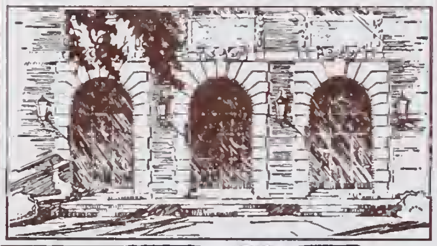


[redacted]

510 S4
il6r

Report No. UIUCDCS-R-76-830

NSF-OCA-MCS73-07980 A03-000024

INTERPROCESSOR CONNECTIONS-- CAPABILITIES, EXPLOITATION AND EFFECTIVENESS

iup

by

Kuo Yen Wen

October 1976

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-76-830

INTERPROCESSOR CONNECTIONS--
CAPABILITIES, EXPLOITATION AND EFFECTIVENESS

by

Kuo Yen Wen

October 1976

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US NSF-MCS73-07980 A03 and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, October 1976.

INTERPROCESSOR CONNECTIONS---
CAPABILITIES, EXPLOITATION AND EFFECTIVENESS

Kuo Yen Wen, PhD
Department of Computer Science
University of Illinois at Urbana-Champaign, 1976

Recently, some research interests has centered around interprocessor connections for SIMD type parallel machines. However, we still lack a methodlogy for evaluating various networks. In this paper, we first present some new results on network properties. Then we show how to exploit various networks in ordinary computations. Finally we describe how we can apply the theoretical results to predict the performance of some network in a real program environment, which is the true measure of network effectiveness.

ACKNOWLEDGMENT

The author is very grateful to Professors Duncan H. Lawrie and David J. Kuck for their constant guidance and suggestions. Special thanks should go to Ross Towle, Bruce Leasure and Mike Wolfe for providing the Analyzer outputs, and to Donald Chang for writing the simulator. Yuzo Hayashi has been very helpful in the course of debugging and in the designing of experiments.

Sincere thanks again go to Donald Chang for drawing some of the illustrations.

TABLE OF CONTENTS

1. INTRODUCTION

Recently, component speeds have been improving at a tremendous rate. Yet there are certain physical limitations to component speeds. Multiprocessing then seems to be the area to show the most promise for any further speedup of computations. The arrival of the cheap but powerful LSI microprocessors greatly increases the attractiveness of multiprocessing systems. However, a big problem arises in finding the best way to interconnect all the processors. The questions that are yet to be answered are what kind of network should we use, how should we compile or restructure computation algorithms in order to use it, and how well does it work on ordinary Fortran programs.

Many interconnection schemes have been proposed or built in recent years. Thurber [1] gives a survey on some of the more important ones. However, each of the networks proposed or built has different requirements to fulfill and their implementations are based on different theoretical backgrounds. Frequently, their capabilities are incompletely known, and their control algorithms are poorly understood. Hence it is very difficult to categorize or assess the merits of each of these networks. Chapter 2 of this thesis investigates the theoretical part of network capabilities. New network properties are presented which help us to utilize certain networks more efficiently. This section also finds some ways to simplify the control algorithms for realizing some commonly used permutations on certain interconnection networks. By simplifying the control algorithms of certain networks, we reduce their network complexity and increase its feasibility.

The attractiveness of a connection network depends not only on its ability to handle some permutations, but also on the efficiency in which some common computations can be mapped into the processing system where it is located. By mapping, we refer to the entire spectrum of strategies including arithmetic operation scheduling, memory storage scheme, and intermediate data routing. It can be shown that a great part of any Fortran program can be represented as either array operations or recurrence systems. So a meaningful processing system will have to be highly efficient in mapping array or recurrence operations into the system. However, it is also desirable to recognize some commonly used computation algorithms, like matrix multiplication and Fast Fourier Transform, and exploit the best execution sequences and mapping strategies in some given processing systems. Chapter 3 deals with the restructuring of certain computation algorithms and the exploitation of certain processor systems.

The true measure of the effectiveness of a processor interconnection scheme (or even a certain parallel computer organization) lies on its performance in a real program environment, not on operation by operation basis, nor on computation by computation basis. A current joint project deals with the simulation of parallel processing systems. Input programs will first be parallelized by a program analyzer. The parallelized program graph is then input into the Resource Request Generator(RRG) which then compile it into pseudo machine code for a specific parallel processing system configuration. The RRG will use most of the known capabilities and properties of interconnection networks and parallel processing systems. Finally the pseudo machine code is simulated on a Simulator which will produce a set of performance measures. This allows us to evaluate various parallel processing system designs upon their performances on real programs. Strategies and algorithms for this work are presented in Chapter 4. Chapter 4 will also include a discussion of some preliminary results.

2. NETWORK CAPABILITIES

2.1 Introduction

In general, processor(/memory) interconnection networks can be divided into two classes. The first class has multiple stages of switching elements. The second class has only a single stage of switching elements and this stage may have to be recycled many times to obtain certain permutations. Examples of the first class are the Batcher network[2], the Benes network[3], the omega network[4], the barrel shifter, and the Feng's data manipulator[5]. Networks such as the Illiac IV connection[6], the Swanson connection[7], the ± 1 shift network and the perfect shuffle network[8] are good examples of the one stage networks. Although single stage networks may be slower in performing general permutations of data than the multistage networks, they are much cheaper in comparison. If we can restructure and recompile some of the commonly used computation algorithms into algorithms which fully utilize the available connectivities, we can retain the performance level while drastically decreasing the cost of the processing system.

Another way of categorizing interconnection network is based on the shuffle connection. Golomb [9], Pease [10], and Stone [8] define the perfect shuffle permutation of a vector of N indices as

$$P(i) = 2i \qquad 0 \leq i < N/2$$
$$= (2i+1) \bmod N \qquad N/2 \leq i < N.$$

let us extend this definition to (x,y) shuffle by defining

$$S(i) = x(i \bmod y) + \lfloor i/y \rfloor \qquad 0 \leq i < N$$

Note that the perfect shuffle is just a (2,N/2) shuffle.

Interconnection networks such as the Batcher network, the Benes network, the omega network, and the Banyan network [11] have stages of switching elements interconnected with shuffle connections. On the other hand, examples of the non-shuffle-based networks are the crossbar switch, the Illiac I^V network, the barrel shifter, the Swanson connection, and Feng's data manipulator.

One of the multiple-stage shuffle-based networks which has been of particular interest to us in recent years is the omega network. This network cannot perform all connections of its inputs to outputs, yet it is capable of producing most of the connections required by numerical programs. Because of the incomplete capabilities of this network, it is necessary to analyze the network to determine exactly which connections it can produce. Section 2.2 will discuss the control structures and some new results on capabilities of the omega network. These results are essential to some of the algorithm solving abilities discussed in Chapter 3.

The relations of the Batcher and Benes networks to the shuffle connections are being discussed in Sections 2.3 and 2.4. Section 2.5 discussed various routing techniques for permutations using one-stage perfect shuffle networks.

2.2 Omega Network

2.2.1 Control Structures for Omega Network

2.2.1.1 Source/Destination Tag Method

The omega network is attractive not only because of its low gate complexity, but also because of its control simplicity. There are a number of ways to control the omega network. The most fundamental method is the destination tag method[4] which uses N destination tags, each of log N bits. Each source port has a tag which represents the destination port numbers the data element intends to reach. log N stages will be required to set the network, and as each stage is set, the data elements will be switched accordingly. This control method can be used to pass all omega-passable permutations.

A more general method is the source tagging method [4]. Instead of using the destination tags, source tags are used. One source tag is associated with each output and represents the input to which that output port will be connected. The source tagging method can pass all omega-passable connections, including the one-to-many-connections that cannot be realized by the destination tag method. However, the main drawback is that the network control has to be set stage by stage from the last stage to the first stage before the data elements can be passed through the network. Hence an extra O(log N) gate delays will be needed for a conection. Details of these two control methods are described in [4].

A modified destination tag method that will allow certain broadcasting (one-to-many) connections will be presented here. Using this modified method, we will do away with the extra $O(\log_2 N)$ gate delays needed for the source tagging method. However, the broadcasting functions will be limited to only one-to-power-of-two elements. For example, the following connection will not be realizable by this method, but can be realized using the source tag method.

Example

Source	Destination
0	0
0	1
0	2
2	3

For this modified tag method, instead of allowing only 0 or 1 for each of the $\log_2 N$ tag bits, we allow '0', '1', '*' or '-' for each of the $\log_2 N$ destination tag characters. So a source/destination pair may now look like:

(0 1 1 0 1, 0 * 1 1 *)

This pair will represent a one-to-four broadcasting function of:

(0 1 1 0 1, 0 0 1 1 0)

(0 1 1 0 1, 0 0 1 1 1)

(0 1 1 0 1, 0 1 1 1 0)

(0 1 1 0 1, 0 1 1 1 1)

The tag character '*' takes on the value of all possible binary digits, while '0' and '1' still have the original meaning. For completeness, we have to use the tag character '-' to specify no connection. So for a complete source/destination set, we might have:

Example

	source port	destination port
(0 0, * 0)	0	0
(0 1, 1 1)	0	2
(1 0, 0 1)	1	3
(1 1, - -)	2	1

It should be clear by now that this modified tag method can only be used if the power of two(say, 2^h) destination ports that a source port is connected to have the same($\log_2 N-h$) bits in their tag bit representations. This modified destination tag method also provides a great notational advantage over the source tag method when we have to describe a one-to-power-of-two broadcasting functions. As can be seen in later chapters, most of the common broadcasting functions are of this type and can be easily described by this modified tag method.

2.2.1.2 Column Control Method

To control an omega network of size N$\times$N for any omega passable permutation, we would require $N\log_2 N/2$ control bits. Each switching element will either be set to the 'cross over' or 'straight through' state according to the value of the corresponding control bit. Suppose we are willing to sacrifice some of the capabilities of the network in order to further simplify the control structure. If we use only $\log_2 N$ control bits, each controlling a complete stage of switching elements, then we will have the column control method. An omega network utilizing column control method turns out to be exactly the same as Batcher's scrambling/unscrambling network[12]. As pointed out in [12], the scrambling/uncrambling network can be constructed with $\log_2 N$ levels of selections and perfect shuffles, just as an omega network.

Let s_{ij} be the jth most significant bit of the bit representation of the ith source port and d_{ij} be the jth most significant bit of the bit representation of the ith destination port. Also let p_j be the jth most significant control bit. Then for any permutation to be realizable by the column control method, $d_{ij} = s_{ij} \oplus p_j$ $\quad \forall j=1\ldots\log_2 N,$
$\forall i=1\ldots.N.$

where $\oplus$ is the exclusive-or operation. More details can be found in [12].

Since there is a total of only $2^{**}(\log_2 N)$ (=N) different sets of p_j's, we can pass at most N distinct permutations using this method. Using the arguments in [12], we can see that if the (i,j)th element of an NxN matrix is stored in the ith position of memory module $i \oplus j$ $(0 \leq i,j < N)$, then we can fetch any row or column of the matrix without conflict and the data can be aligned using this column control method. Hence if column and row accessings are all that are required, we can simply use a very easily controlled column control method for our omega network.

It is logical to think that by increasing the number of shuffle-exchange stages, we would be able to obtain more permutations that can utlilize this column control method. Unfortunately, it can be shown that by increasing the number of shuffle-exchange stages, we can only derive the original set of permutations plus the 1-shuffled, 2-shuffled, ...$(\log_2 N-1)$-shuffled versions of the permutations. Hence the maximum number of this set does not exceed $N\log_2 N$. The proof will not be elaborated here, but Figure 2.1 should illustrate this phenomenon. Suppose that we want to have a five-stage network using this column control method (either by building five stages of shuffle-exchanges or recycling a one stage network five times), and also that we set p to 01111. . The permutation that we get at the output will be (5 7 1 3 4 6 0 2) for an 8x8 network. However, the fourth most significant bit of p will force a shuffle and then 'exclusive or' with the most significant bit of the source

	S	S	E	S	E	S	E	S	E
0	0	0	2	2	3	3	7	7	5
1	4	2	0	3	2	7	3	5	7
2	1	4	6	0	1	2	6	3	1
3	5	6	4	1	0	6	2	1	3
4	2	1	3	6	7	1	5	6	4
5	6	3	1	7	6	5	1	4	6
6	3	5	7	4	5	0	4	2	0
7	7	7	5	5	4	4	0	0	2

Figure 2.1.1 Intermediate patterns using p=01111

	S	E	S	S	E	S	S
0	0	4	4	4	5	5	5
1	4	0	6	5	4	1	7
2	1	5	0	6	7	4	1
3	5	1	2	7	6	0	3
4	2	6	5	0	1	7	4
5	6	2	7	1	0	3	6
6	3	7	1	2	3	6	0
7	7	3	3	3	2	2	2

Figure 2.1.2 Intermediate patterns using $p=011 \oplus 110=101$

plus two shuffles at the end

Figure 2.1

tags again. A similar effect will be produced by the least significant bit of p on the second most significant bit of the source tags. The net effect will then be equivalent to setting p=011$\oplus$110(=101) and adding two extra shuffles at the end. Note that the output of Figure 2.1.2 is also (5 7 1 3 4 6 0 2). So by setting p=01111, we only result in the 2-shuffled version of setting p=101. However, it should be noted that the Nlog N upper limit on number of passable permutations only applies to the column control method. For any individual switch control method (such as the source/destination tag method and the ROM method), the upper limit depends on the number of stages.

The permutation capabilities of the $\log_2 N$ stage column controlled omega network are well defined by [12], and we will not repeat his results here. However, we can increase the capabilities of a column controlled network to allow certain broadcasting functions by using two control bits, b_0 and b_1, in each column. In Chapter 3, it will be shown what broadcasting functions can be realized by this method. The switching functions for various values of b_0 and b_1 are shown in Figure 2.2.

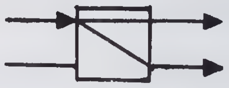
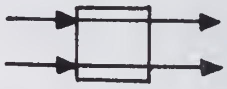
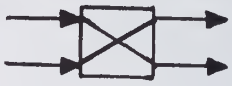
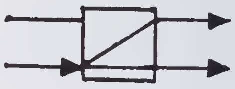
b_0	b_1	Action	Illustration
0	0	upper broadcast	
0	1	straight pass	
1	0	cross over	
1	1	lower broadcast	

Figure 2.2

2.2.1.3 ROM Control Method

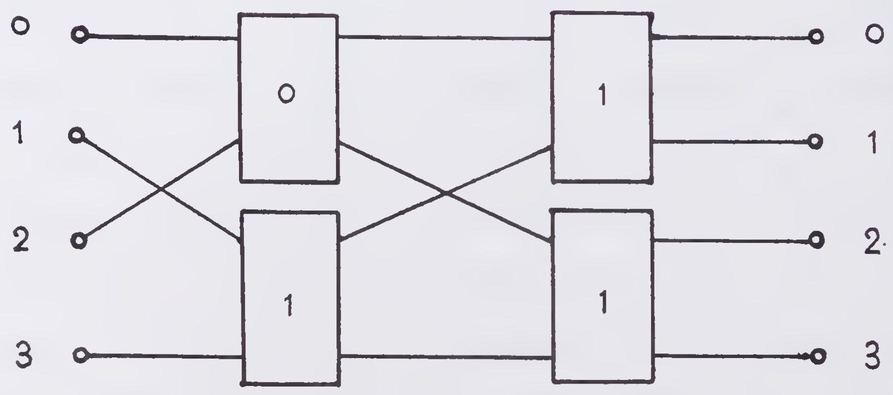
To implement the source/destination tag method, we either use a fast method [13] which would require $8N\log_2 N(d+(\log_2 N-1)/2)$ gates, or a slower method [14] which needs $(4d+11)N\log_2 N$ gates but requires the use of strobes at each stage to pass the tag bits along. The column control method also needs the use of strobes if a one-stage shuffle-exchange network is used. So the propagation delay through the network is on the order of $\log_2 N$ clocks. In this section, we will propose another control method which can eliminate the use of the strobes(clockings) without paying too much penalty in gate counts. This ROM control method provides a faster method to evaluate the control function at each of the $N\log_2 N/2$ switches simultaneously and these functions are imposed on all stages of the network at the same time. So instead of taking $\log_2 N$ clocks through the network, we would only require a couple of clocks for the source data to be routed through. This greatly reduces the network delay for the processing system. This method does not pass as many permutations as the source/destination tag method, but it will pass many of the more common ones. It car pass all shift, flip, (c-i) , and odd-ordered vector unscrambling permutations in any power of two partitions.

We again assume, as in Section 2.2.1.2, that a control bit value of 1 will set a switching element to the 'cross over' state and the value of 0 will set it to the 'straight through' state. The basic idea is to fetch N/2 control bits from each of $\log_2 N$ ROM's, according to which permutation function is called for. The array of $\log_2 N \times N/2$ control bits is called the control matrix and can be imposed on the omega network to facilitate the corresponding permutation function.

For example, the control matrix for a 1-shift permutation in a 4x4 omega network is:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

Imposing it on an omega network, we get:



We will immediately get the following permutation :

0 ----> 1 ,which is a 1-shift permutation.

1 ----> 2

2 ----> 3

3 ----> 0

In order to minimize the amount of ROM space required for different families of permutations, as many common characteristics are recognized from the control patterns as possible. Then, by using some extra logical operations, we can form the same control pattern with much less ROM space required. We first observe that the flip permutation actually belongs to the family of (c-i) permutations, with c set to N-1. Then we observe that the control matrix for (c-i) permutation is the bit by bit complement of the control matrix for a (N-1-c) shift permutation. Note that (N-1-c) is the bit complement of the bit representation of c.

Example: For N=8, the control matrices are:

For 5-shift: $\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$ For (2-i) perm.: $\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$

With this knowledge, we can already reduce the three classes of permutations(shift, (c-i), and flip) into one set of control patterns.

We further observe that for any permutation to be done in a smaller partition(say, 2^h), we only need to use the same control matrix as for the same permutation in the full network, with the exception that the leftmost ($\log_2 N-h$) columns will be set to all 0's. This fact enables us to greatly reduce the number of control matrices for all the partitioned permutations by simply using a small number of AND gates controlled by the partition size.

Figure 2.3 shows the control matrices for both the 3-shift permutation in full 8x8 network and the 3-shift permutation in 4-partition of an 8x8 network.

$$\begin{array}{cc} \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} & \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \\ \text{3-shift} & \text{3-shift in 4-partition} \end{array}$$

Figure 2.3 Control Matrices for 3-Shift in different partitions

Note that the only difference between the two matrices is that the second one has all 0's in the first column.

Even with these control techniques, the amount of ROM space required for all the shift patterns are still high. There are N different shifts and each requires $N\log_2 N/2$ bits. So a total of $N^2\log_2 N/2$ bits will be required. However, we can extract more information from the shift patterns to reduce the total ROM space down to $N^2/2$ bits, which greatly increases the feasibility of this control method. The control patterns for various shift permutations for N=8 are shown in Figure 2.4. Let $n=\log_2 N$ and $s_1 s_2 \ldots s_n$ be the bit representation of the shift distance. In general, there are N different patterns for the leftmost column. However, only N/2 of them are the basic patterns, and shift distances with the same $s_2 s_3 \ldots s_n$ will have the same basic patterns. Hence $s_2 s_3 \ldots s_n$ can be used as the address to access the corresponding pattern stored in the ROM. The ROM for this first column will have N/2 entries of N/2 bits. The exact pattern will be the 'exclusive or' of s_1 independently with the column of control bits. For the second column of the control matrices, there are N/4 different basic patterns and shift distances with the same $s_3 s_4 \ldots s_n$ will have the same basic patterns. The exact pattern will be the 'exclusive or' of s_2 with the column of bits. The number of basic patterns decrease from left to right. The last column will have only one basic pattern and will be exclusive-ored with s_n to form the correct patterns. The total number of basic patterns required for the shift, (c-i) and flip permutations in any power of two partition can then be realized using a total ROM

shift	control			shift	control		
distance	pattern			distance	pattern		
0	0	0	0	4	1	0	0
	0	0	0		1	0	0
	0	0	0		1	0	0
	0	0	0		1	0	0
1	0	0	1	5	1	0	1
	0	0	1		1	0	1
	0	1	1		1	1	1
	1	1	1		0	1	1
2	0	1	0	6	1	1	0
	0	1	0		1	1	0
	1	1	0		0	1	0
	1	1	0		0	1	0
3	0	1	1	7	1	1	1
	1	1	1		0	1	1
	1	0	1		0	0	1
	1	0	1		0	0	1

Figure 2.4 Control Patterns for Shift Permutations

$N=8$

space of (1+2+4+...N/2) bits =N(N-1)/2 bits, with the help of some additional logic elements.

A similar phenomenon to that in the shift patterns can be seen for odd-ordered vector unscrambling permutations. The control patterns for various odd-ordered vector unscrambling for N=16 are shown in Figure 2.5. Let $p_1 p_2 \cdots p_n$ be the bit representation of the order of unscrambling. Then $p_1 p_2 \cdots p_{n-1}$ will be used as an address to fetch the basic pattern for the first column, $p_2 \cdots p_{n-1}$ as the address to fetch the pattern for the second column, and so on. The output, however, does not need to be exclusive-ored with p_i (as in the case of shift patterns) to produce the correct patterns.

A possible organization of the control system is shown in Figure 2.6. Using microprogramming, we can set $k_1 k_2 \cdots k_n$ to $s_1 s_2 \cdots s_n$ for shifts, to $\bar{c}_1 \bar{c}_2 \cdots \bar{c}_n$ for (c-i) permutations, and to $0p_1 p_2 \cdots p_{n-1}$ for odd-ordered vector unscrambling.

The basic control patterns have to be generated and input into the ROM's. The basic shift patterns can be generated quite easily. There will be $N/2^j$ entries in the jth ROM from the left($1 \leq j \leq \log_2 N$). The 0th entry in each of the ROM's will have all 1's. For the kth entry ($1 \leq k < N/2^j$) in the jth ROM, the least significant ($k.2^{j-1}$) control bits will be 1's, while the rest are all 0's. To generate the

order	control pattern				order	control pattern			
1	0	0	0	0	9	0	0	0	0
	0	0	0	0		1	0	0	0
	0	0	0	0		0	0	0	0
	0	0	0	0		1	0	0	0
	0	0	0	0		0	0	0	0
	0	0	0	0		1	0	0	0
	0	0	0	0		0	0	0	0
	0	0	0	0		1	0	0	0
3	0	0	0	0	11	0	0	0	0
	0	0	0	0		1	0	0	0
	0	0	0	0		0	0	0	0
	1	0	0	0		0	0	0	0
	1	1	1	0		1	1	1	0
	1	1	1	0		0	1	1	0
	0	0	1	0		0	0	1	0
	0	0	1	0		1	0	1	0
5	0	0	0	0	13	0	0	0	0
	0	0	0	0		1	0	0	0
	1	1	0	0		1	1	0	0
	1	1	0	0		0	1	0	0
	0	0	0	0		0	0	0	0
	1	0	0	0		0	0	0	0
	1	1	0	0		1	1	0	0
	0	1	0	0		1	1	0	0
7	0	0	0	0	15	0	0	0	0
	0	0	0	0		1	0	0	0
	1	1	0	0		1	1	0	0
	0	1	0	0		1	1	0	0
	1	1	1	0		1	1	1	0
	0	1	1	0		1	1	1	0
	1	1	1	0		1	1	1	0
	0	1	1	0		1	1	1	0

Figure 2.5 Control Patterns for p-unscrambling

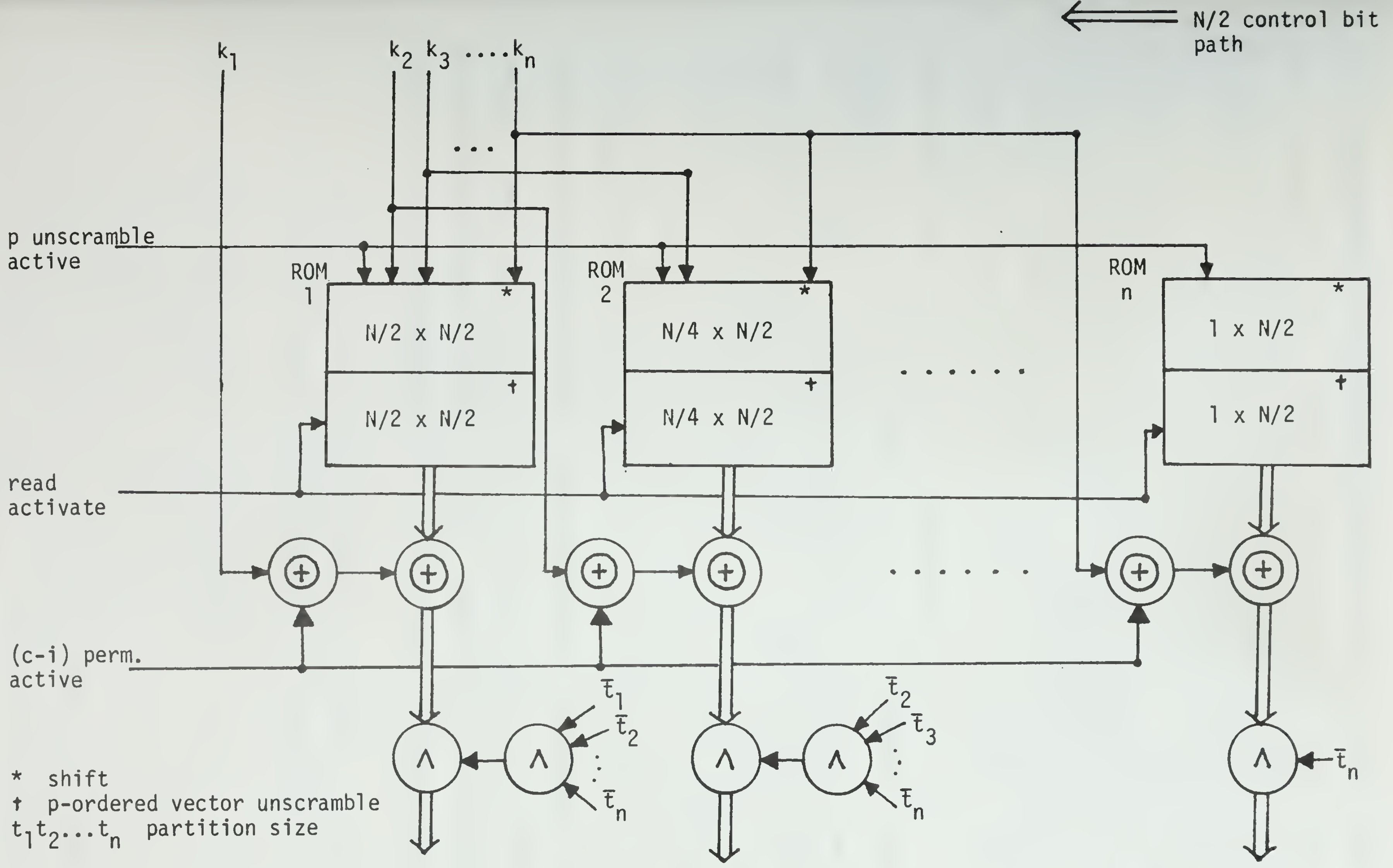


Figure 2.6 An Omega ROM Control Circuit

p-ordered vector unscrambling patterns, it requires a bit more work. For the first stage pattern, the ith control bit ($0 \leq i < N/2$) for p-unscrambling(i.e. the $(p-1)/2$ th entry in the RCM), is the ($\log_2 N$)th least significant bit of the product ($p.i$). For the jth stage pattern ($j \neq 1$), the ith control bit ($0 \leq i < N$) in the kth entry ($0 \leq k < N/2$) will be the same as the ($\lfloor i/2^{j-1} \rfloor * 2^{j-1}$)th control bit in the kth entry of the first stage RCM.

This section indicates how the use of ROM's can eliminate the need to clock the omega network, at the expense of some relatively inexpensive hardware. The set of allowable permutations is quite rich. Hence the ROM method should be considered as a major alternative to the source/destination tag method.

It should be pointed out here that an alternative to the ROM may be the use of an initial set of control bits that are being routed through the network, but with some logical operations at each stage. This method has been discussed by Lang and Stone[15], and will not be repeated here.

2.2.2 Omega Partition Theorems

One important property of the omega network is its ability to be partitioned. The theorems in this section will show that a large size omega network can be regarded as a conglomeration of many smaller size omega networks, each passing a different smaller omega-passable connection function. These partition theorems help to establish many capabilities of a larger size network on smaller partition connections.

Example

Given an 8x8 omega network. Assume that source ports 0-3 want to do an end-around 1-shift. Assume also that destination ports 4-7 request data from source port 5. So the complete set of source destination pairs is P = {(0,1),(1,2),(2,3),(3,0),(5,4),(5,5),(5,6),(5,7)}. We know that a 4x4 omega network can perform an end-around 1-shift, as well as a one-to-many broadcasting function. By using the partition theorem stated below, we can be sure that an 8x8 omega network can pass P.

Before we state the partition theorems, we would like to list, without proofs, some number theory results.

R0) $a \underset{m}{\equiv} b,\ c \underset{m}{\equiv} d \ \text{-->}\ ax+cy \underset{m}{\equiv} bx+dy$

R1) $x+a \underset{m}{\equiv} y+a \ \text{-->}\ x \underset{m}{\equiv} y$

R2) $x \underset{m}{\equiv} y \ \text{-->}\ ax \underset{m}{\equiv} ay$

R3) If a is prime to m (i.e. gcd(a,m)=1), then

$ax \underset{m}{\equiv} ay \ \text{-->}\ x \underset{m}{\equiv} y$

R4) $ax \underset{am}{\equiv} ay \longleftrightarrow x \underset{m}{\equiv} y$

R5) If $0 \leq x, y < m$, then

$x \underset{am}{\equiv} y \longleftrightarrow x \underset{m}{\equiv} y$

R6) $x \underset{m}{\equiv} y$ and $x \underset{N}{\not\equiv} y \longrightarrow x \overset{m}{\underset{N}{\not\equiv}} y$

R1 through R6 can be found in [4].

We will also present Lemma 2.1 which is extended from Lemma 1 of [4] and some of the above number theory results.

Lemma 2.1

Let $0 \leq x_1, y_1 \leq n-1$, and $0 \leq x_2, y_2 \leq a-1$. Then $ax_1 + x_2 \underset{an}{\equiv} ay_1 + y_2 \longleftrightarrow x_1 = y_1$ and $x_2 = y_2$.

Proof:

a) to prove $ax_1 + x_2 \underset{an}{\equiv} ay_1 + y_2 \longrightarrow x_1 = y_1$ and $x_2 = y_2$: proved in Lemma 1 of [4].

b) to prove $x_1 = y_1$ and $x_2 = y_2 \longrightarrow ax_1 + x_2 \underset{an}{\equiv} ay_1 + y_2$: assume $x_1 = y_1$ and $x_2 = y_2$.

By R4, $ax_1 \underset{an}{\equiv} ay_1$. Since $0 \leq x_2, y_2 < a$, then by R5, we have $x_2 \underset{an}{\equiv} y_2$. So by R0, we get $ax_1 + x_2 \underset{an}{\equiv} ay_1 + y_2$, QED.

To simplify the proof of the partition theorems, we need to restate Theorem 2 in [4], which dictates whether a given connection is omega-passable or not.

Lemma 2.2 (Equivalent Statement of Theorem 2 in [4])

Given a set of desired input-output connection $P_N = \{(S_i, D_i) \mid 0 \leq i < N\}$, then an NxN omega passes P_N if and only if for all S-D pairs in P_N and for all $m = 2^k$, where $1 \leq k \leq \log_2 N$, $S_i \underset{N}{\equiv} S_j$ or $S_i \underset{m}{\not\equiv} S_j$ or $D_i \overset{m}{\underset{N}{\not\equiv}} D_j$.

Definition 2.1

Let $P_L=\{(s_i,d_i)\,|\,0\leq i<L\}$ and $P_{M_i}=\{(t_{ij},e_{ij})\,|\,0\leq j<M\}$, $0\leq i<L$. We define $P_N=P_L\times\{P_{M_0},P_{M_1},\ldots\ldots P_{M_{L-1}}\}$

$$=\{(s_iM+t_{ij},d_iM+e_{ij})\,|\,0\leq i<L,\ 0\leq j<M\}$$

For example, let $L=4$, $M=4$ and $N=16$. If

$P_{M_0}=\{(0,1),(1,2),(2,3),(3,0)\}$, a 1-shift permutation,

$P_{M_1}=\{(0,0),(0,1),(0,2),(0,3)\}$, a 1-to-4 broadcast connection,

$P_{M_2}=\{(0,3),(1,2),(2,1),(3,0)\}$, a flip permutation,

$P_{M_3}=\{(0,0),(1,3),(2,2),(3,1)\}$, a 3-order unscrambling, and

$P_L=\{(0,2),(1,3),(2,0),(3,1)\}$, a 2-shift permutation.

Then by definition, $P_N=\{(0,9),(1,10),(2,11),(3,8),(4,12),(4,13),(4,14),(4,15),(8,3),(9,2),(10,1),(11,0),(12,4),(13,7),(14,6),(15,5)\}$.

In words, the sources and destinations of P_N are divided into 4 partitions. P_L is the inter-partition permutation function, and P_{M_i}'s are the individual partition permutations. P_L moves partition #0 to partition #2 and then the individual elements in partition #2 will be moved according to P_{M_0}, and so on. A pictorial illustration of P_N is shown in Figure 2.7.

With all these preliminary definitions and lemmas, we can present the first of the omega partition theorems.

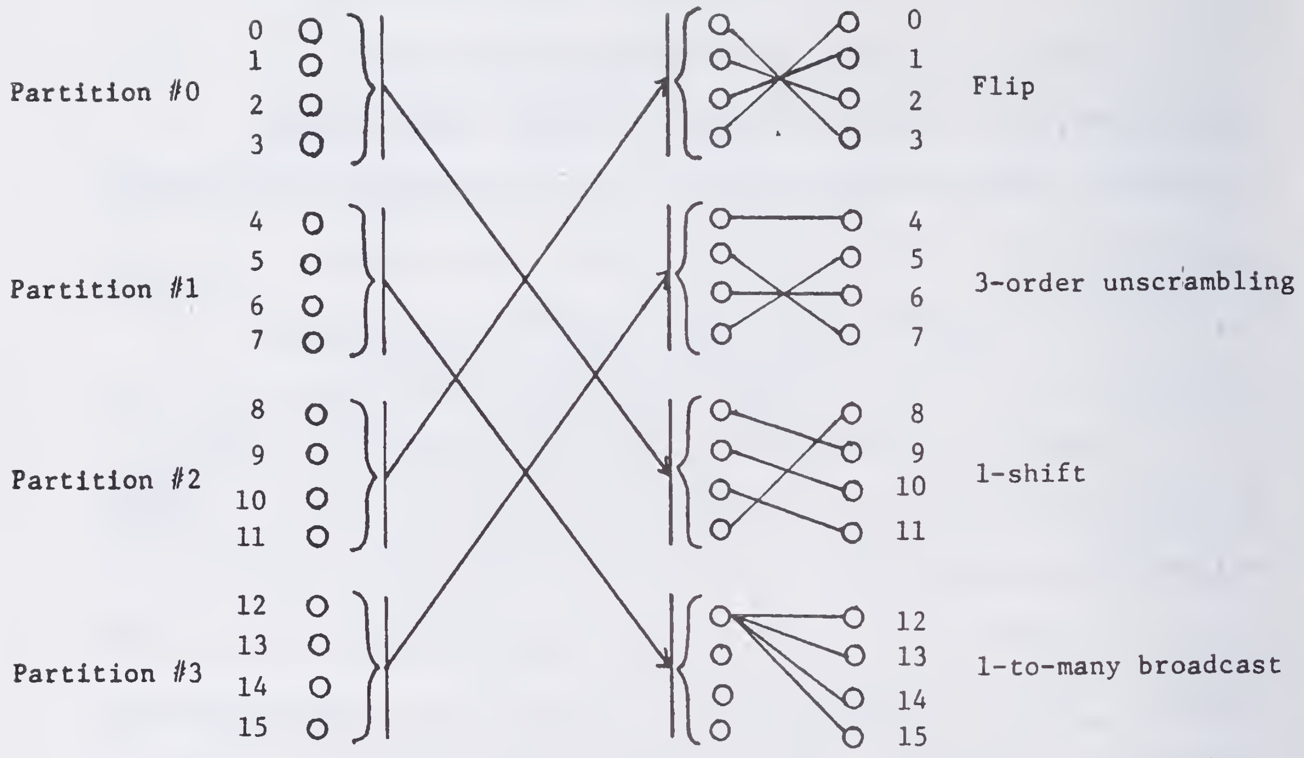


Figure 2.7 A Partitioned Permutation

Theorem 2.1

Let $\Omega_L \uparrow P_L$ and $\Omega_M \uparrow P_{M_i}$, $0 \leq i \leq L$, and let $N = L \times M$, then $\Omega_N \uparrow P_N = P_L \times \{P_0, P_1, \ldots, P_{L-1}\}$.

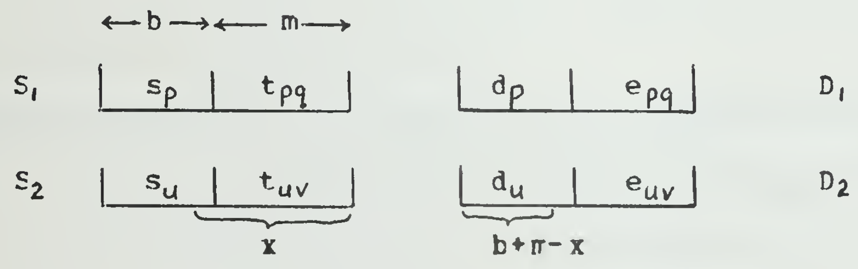
We first present a simple sketch proof in Proof 1. then we present a more rigorous proof in Proof 2.

Proof 1

Assume $\Omega_N \uparrow P_N$. By Lemma 2.2, there exist $S_1 = s_p M + t_{pq}$, $D_1 = d_p M + e_{pq}$, $S_2 = s_u M + t_{uv}$, $D_2 = d_u M + e_{uv}$ and X such that $S_1 \underset{N}{\not\equiv} S_2$ and $D_1 \overset{X}{\underset{N}{\equiv}} D_2$.

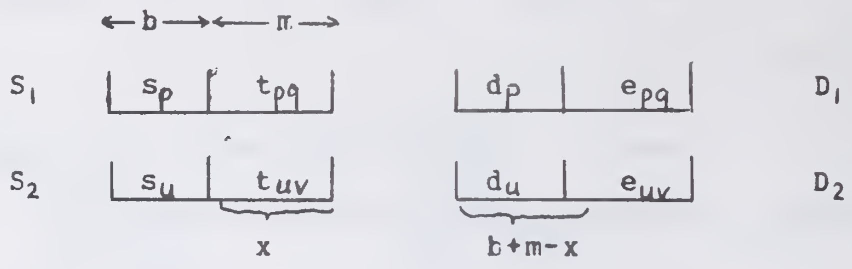
Let m=logM, n=logN, b=logL, and x=logX.

It $X \geq M$, pictorially we have:



Here the trailing x bits of S_1 and S_2 are equal, but the leading $(b+m-x)$ bits are not equal, and the leading $(b+m-x)$ bits of D_1 and D_2 are equal. Since $x \leq m$, the trailing $(a=x-m)$ bits of s_p and s_u are equal but the leading $(b-a)$ bits are different and the leading $(b-a)$ bits of d_p and d_u are equal. This contradicts $\Omega_L \uparrow P_L$.

If $X < M$, pictorially we have:



Since the leading (b+m-x) bits of D_1 and D_2 are equal and $m>x$, we have $d_p=d_u$ and $e_{pq} \overset{X}{\underset{M}{\equiv}} e_{uv}$. Since $\Omega_L \uparrow P_L$ and $d_p=d_u$, p has to be equal to u. So $s_p=s_u$. This implies that $t_{pq}=t_{uv}$ since $S_1 \underset{N}{\not\equiv} S_2$. $S_1 \underset{X}{\equiv} S_2$ and $X<M$ imply that $t_{pq} \underset{X}{\equiv} t_{uv}$. Setting $p=u$, we get $e_{pq} \overset{X}{\underset{M}{\equiv}} e_{pv}$, $t_{pq} \underset{M}{\not\equiv} t_{pv}$ and $t_{pq} \underset{X}{\equiv} t_{pv}$. They imply that $\Omega_M \uparrow P_{M_p}$, which is a contradiction. Hence, Theorem 2.1 is proved.

Proof 2

Assume $\Omega_N \uparrow P_N$, then by Lemma 2.2, $\exists (s_pM+t_{pq}, d_pM+e_{pq})$ and $(s_uM+t_{uv}, d_uM+e_{uv})$ and $X=2^x$ such that:

(a) $s_pM+t_{pq} \underset{N}{\not\equiv} s_uM+t_{uv}$

and (b) $s_pM+t_{pq} \underset{X}{\equiv} s_uM+t_{uv}$

and (c) $d_pM+e_{pq} \overset{X}{\underset{N}{\equiv}} d_uM+e_{uv}$

By Lemma 2.1 and (a), $s_p \underset{L}{\not\equiv} s_u$ or $t_{pq} \underset{M}{\not\equiv} t_{uv}$ (2.1)

If $X \geq M$, let $A=X/M=2^a$.

By Lemma 2.1 and (b), $s_p \underset{A}{\equiv} s_u$ and $t_{pq} \underset{M}{\equiv} t_{uv}$ (2.2)

From (2.1) and (2.2) we get $s_p \overset{A}{\underset{L}{\not\equiv}} s_u$ (2.3)

and $s_p \underset{A}{\equiv} s_u$ (2.4)

By definition, (c) implies $X\left\lfloor\frac{d_p M+e_{uv}}{X}\right\rfloor \underset{N}{\equiv} X\left\lfloor\frac{d_u M+e_{uv}}{X}\right\rfloor$

$\Longrightarrow\quad X\left\lfloor\frac{d_p M}{X}\right\rfloor \underset{N}{\equiv} X\left\lfloor\frac{d_u M}{X}\right\rfloor$ since $e_{pq}, e_{uv} < M \le X$

$\Longrightarrow\quad MA\lfloor d_p/A\rfloor \underset{LM}{\equiv} MA\lfloor d_u/A\rfloor$

$\Longrightarrow\quad A\lfloor d_p/A\rfloor \underset{L}{\equiv} A\lfloor d_u/A\rfloor$ by R4

$\Longrightarrow\quad d_p \overset{A}{\underset{L}{\equiv}} d_u$ (2.5)

(2.3), (2.4) and (2.5) imply $\Omega_L \uparrow P_L$, by Lemma 2.2, contradiction.

If $X<M$, let $B=M/X$.

(b) $\Longrightarrow\quad s_p BX+t_{pq} \underset{X}{\equiv} s_u BX+t_{uv}$

$\Longrightarrow\quad t_{pq} \underset{X}{\equiv} t_{uv}$ (2.6), since $s_p BX, s_u BX \underset{X}{\equiv} 0$

(c) $\Longrightarrow X\left\lfloor\frac{d_p M+e_{pq}}{X}\right\rfloor \underset{N}{\equiv} X\left\lfloor\frac{d_u M+e_{uv}}{X}\right\rfloor$

$\Longrightarrow\quad Xd_p B+X\lfloor e_{pq}/X\rfloor \underset{N}{\equiv} Xd_u B+X\lfloor e_{uv}/X\rfloor$ since M/X is integer

$\Longrightarrow\quad d_p M+X\lfloor e_{pq}/X\rfloor \underset{LM}{\equiv} d_u M+X\lfloor e_{uv}/X\rfloor$

$\Longrightarrow\quad d_p \underset{L}{\equiv} d_u$ and $X\lfloor e_{pq}/X\rfloor \underset{M}{\equiv} X\lfloor e_{uv}/X\rfloor$ by Lemma 2.1

and since $0 \le d_p, d_u < L$, and $0 \le X\lfloor e_{pq}/X\rfloor \le e_{pq} < M$,

and $0 \le X\lfloor e_{uv}/x\rfloor \le e_{uv} < M$.

$\Longrightarrow\quad d_p \underset{L}{\equiv} d_u$ (2.7)

and $\quad e_{pq} \overset{X}{\underset{M}{\equiv}} e_{uv}$ (2.8)

For (2.7) to be true and not contradicting $\Omega_L \uparrow P_L$,

$p = u$ (2.9)

Obviously $s_p \underset{L}{\equiv} s_u$ and therefore (2.1) implies $t_{pq} \underset{M}{\not\equiv} t_{uv}$ (2.10)

Rewriting (2.10), (2.6) and (2.8) setting p=u, we get:

$t_{pq} \underset{M}{\not\equiv} t_{pv}$ (2.11)

$$t_{pq} \underset{x}{\equiv} t_{pv} \qquad (2.12)$$

and $$\epsilon_{pq} \overset{x}{\underset{M}{\equiv}} e_{pv} \qquad (2.13)$$

(2.11), (2.12) and (2.13) imply $\Omega_M \not\uparrow P_{M_p}$, contradiciton.

Hence $\Omega_N \uparrow P_N$ Q.E.D.

The following corollary is a direct consequence of Theorem 2.1.

<u>Corollary 2.1</u>

An NxN omega network can be made to behave as N/M independent MxM omega networks.

<u>Proof</u>:

Let $L=N/M$ and $P=\{(i,i) \mid 0 \le i < \}$, then $\Omega_L \uparrow P_L$ trivially. Hence it we can pick N/M P_M's $\{P_{M_0}, P_{M_1}, \ldots, P_{M_{L-1}}\}$ that are omega passable, then, by Theorem 2.1, $\Omega_N \uparrow P_N = P_L \times \{P_{M_0}, P_{M_1}, \ldots, P_{M_{L-1}}\}$.

In Theorem 2.1, the tag bits denoting the partitioning are the most significant $\log_2 L$ bits. In the following two theorems, we extend the result to any set of $\log L$ bits in the tag representation. The two theorems will not be proved formally. However, the illustrations could be easily generalized to formal proofs.

Let us look at (s_i, d_i) and (s_j, d_j) like the following:

(s_i, d_i): $(\underline{x}_1 x_2 x_3 \underline{x}_4 x_5 x_6 x_7 \underline{x}_8 x_9 \underline{x}_{10}, \underline{y}_1 y_2 y_3 \underline{y}_4 y_5 y_6 y_7 \underline{y}_8 y_9 \underline{y}_{10})$

(s_j, d_j): $(\underline{a}_1 a_2 a_3 \underline{a}_4 a_5 a_6 a_7 \underline{a}_8 a_9 \underline{a}_{10}, \underline{b}_1 b_2 b_3 \underline{b}_4 b_5 b_6 b_7 \underline{b}_8 b_9 \underline{b}_{10})$

Assume there are $\log_2 L$ underlined bits and $\log M$ non-underlined bits.

Theorem 2.2

Assume all the underlined bits of (s,d) satisfy an omega passable connection[†] P_L, and all the non-underlined bits of (s,d) satisfy an omega passable connection P_{M_k}, where k represents the total numerical value of the underlined bits of $\underline{s}$. Then $\{(s,d)\}$ is passable by Ω_{LM}.

Proof:

Assume $\{(s,d)\}$ is not passable by Ω_{LM}, then there

---

† Note that a connection can be a broadcasting function, while a permutation cannot.

exist (s_i, d_i) , (s_j, d_j) ,and r (say, =6, in this case) such

that $\underline{x}_1 x_2 x_3 \underline{x}_4 \not\equiv \underline{a}_1 a_2 a_3 \underline{a}_4$

and $x_5 x_6 x_7 \underline{x}_8 x_9 \underline{x}_{10} \equiv a_5 a_6 a_7 \underline{a}_8 a_9 \underline{a}_{10}$

and $\underline{y}_1 y_2 y_3 \underline{y}_4 \equiv \underline{b}_1 b_2 b_3 \underline{b}_4$

Case 1: $\underline{x}_1 \underline{x}_4 \not\equiv \underline{a}_1 \underline{a}_4$

In this case, we have $\underline{x}_8 \underline{x}_{10} \equiv \underline{a}_8 \underline{a}_{10}$, and $\underline{x}_1 \underline{x}_4 \not\equiv \underline{a}_1 \underline{a}_4$, and $\underline{y}_1 \underline{y}_4 \equiv \underline{b}_1 \underline{b}_4$, which implies $\Omega_L \not\uparrow P_L$, a contradiction.

Case 2: $\underline{x}_1 \underline{x}_4 \equiv \underline{a}_1 \underline{a}_4$

Here we have $\underline{x}_1 \underline{x}_4 \underline{x}_8 \underline{x}_{10} \equiv \underline{a}_1 \underline{a}_4 \underline{a}_8 \underline{a}_{10}$, and s_i and s_j will have the same k value. Now $x_2 x_3 \not\equiv a_2 a_3$ and $x_5 x_6 x_7 x_9 \equiv a_5 a_6 a_7 a_9$ and $y_2 y_3 \equiv b_2 b_3$, which contradict $\Omega_M \uparrow P_{M_K}$.

Hence Theorem 2.2 is proved by contradiction.

Thoerem 2.3

Assume all the underlined bits of (s,d) satisfy an omega passable permutation P_L, and all the non-underlined bits of (s,d) satisfy an omega passable connection P_{M_K}, where k represents the total numerical value of the underlined bits of $\underline{d}$. Then {(s,d)} is passable by Ω_{LM}.

Proof:

Assume {(s,d)} is not passable by Ω_{LM} , then there exist (s_i, d_i) , (s_j, d_j) ,and r (say, =6, in this case) such

that $\underline{x}_1 x_2 x_3 \underline{x}_4 \not\equiv \underline{a}_1 a_2 a_3 \underline{a}_4$,

and $x_5 x_6 x_7 \underline{x}_8 x_9 \underline{x}_{10} \equiv a_5 a_6 a_7 \underline{a}_8 a_9 \underline{a}_{10}$, and $\underline{y}_1 y_2 y_3 \underline{y}_4 \equiv \underline{b}_1 b_2 b_3 \underline{b}_4$.

Case 1: $\underline{x}_1 \underline{x}_4 \not\equiv \underline{a}_1 \underline{a}_4$

In this case, we have $\underline{x}_1 \underline{x}_4 \not\equiv \underline{a}_1 \underline{a}_4$, and $\underline{x}_8 \underline{x}_{10} \equiv \underline{a}_8 \underline{a}_{10}$, and $\underline{y}_1 \underline{y}_4 \equiv \underline{b}_1 \underline{b}_4$, which imply $\Omega_L \not\updownarrow P_L$, a contradiction.

Case 2: $\underline{x}_1 \underline{x}_4 \equiv \underline{a}_1 \underline{a}_4$

This implies that $x_2 x_3 \not\equiv a_2 a_3$

Also we will get $\underline{x}_1 \underline{x}_4 \underline{x}_8 \underline{x}_{10} \equiv \underline{a}_1 \underline{a}_4 \underline{a}_8 \underline{a}_{10}$. Since P_L is a permutation, $\underline{y}_1 \underline{y}_4 \underline{y}_8 \underline{y}_{10} \equiv \underline{b}_1 \underline{b}_4 \underline{b}_8 \underline{b}_{10}$. Hence d_i and d_j will have the same k value. Now $x_2 x_3 \not\equiv a_2 a_3$, and $x_5 x_6 x_7 x_9 \equiv a_5 a_6 a_7 a_8$, and $y_2 y_3 \equiv b_2 b_3$, which contradicts with $\Omega_M \uparrow P_{M_K}$.

Hence Theorem 2.3 is proved.

It should be noted while basically all three theorems allow P_{M_K}'s to be any connection function, P_L in Theorem 2.3 is the only one that requires to be a permutation. To help appreciate why P_L has to be a permutation in Theorem 2.3, but not in Thoerem 2.1, we will show the following example.

<u>Example</u>:

For L=2, M=4, we let P be {(0,0), (0,1)}, P_{M_0} be a 1-shift permutation and P_{M_1} be a 2-shift permutation.

1) Let the most significant bit be the underlined bit, we have:

$\underline{s}_1$	s_2	s_3	$\underline{d}_1$	d_2	d_3
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	0	0	0
0	0	0	1	1	0
0	0	1	1	1	1
0	1	0	1	0	0
0	1	1	1	0	1

There is no conflict.

2) Let the middle bit be the underlined bit, we have:

s_1	$\underline{s}_2$	s_3	d_1	$\underline{d}_2$	d_3
0	0	0	0	0	1
0	0	1	1	0	0
1	0	0	1	0	1
1	0	1	0	0	0
0	0	0	1	1	0
0	0	1	1	1	1
1	0	0	0	1	0
1	0	1	0	1	1

There are many conflicts(e.g., between (0,1) and (4,2)).

3) Let the least significant bit be the underlined bit, we have:

s_1	s_2	$\underline{s}_3$	d_1	d_2	$\underline{d}_3$
0	0	0	0	1	0
0	1	0	1	0	0
1	0	0	1	1	0
1	1	0	0	0	0
0	0	0	1	0	1
0	1	0	1	1	1
1	0	0	0	0	1
1	1	0	0	1	1

There are many conflicts(e.g., between (0,2) and (4,1)).

This partitioning property of the omega network proves to be vital for the efficient handling of many algorithms, especially the Recurrence Solvers, as shown in Chapter 3 of this thesis.

2.2.3 Omega Broadcast Theorems

Theorems 10 and 11 of [4] describe broadcasting functions for square blocks. In this section, we are going to extend them to 3-dimensional arrays, not necessary of equal size edges.

We use the notation $(k,x,y)\langle a,b,c\rangle$ to denote element (k,x,y) of an axbxc array. Here $0\le k<a$, $0\le x<b$, $0\le y<c$. Also $(k,x,y)\langle a,b,c\rangle \dashrightarrow (*,x,y)\langle a,b,c\rangle$ symbolizes the mapping of element (k,x,y) to elements $(0,x,y),(1,x,y),\ldots(a-1,x,y)$.

Now we can show six extensions of the broadcast theorems.

For constant k and for all values of x,y,*,

(i) $\Omega_{abc} \uparrow \{(k,x,y)\langle a,b,c\rangle \dashrightarrow (*,x,y)\langle a,b,c\rangle\}$

(ii) $\Omega_{abc} \uparrow \{(k,x,y)\langle a,b,c\rangle \dashrightarrow (x,*,y)\langle b,a,c\rangle\}$

(iii) $\Omega_{abc} \uparrow \{(k,x,y)\langle a,b,c\rangle \dashrightarrow (x,y,*)\langle b,c,a\rangle\}$

(iv) $\Omega_{abc} \uparrow \{(k,x,y)\langle a,b,c\rangle \dashrightarrow (y,x,*)\langle c,b,a\rangle\}$ iff $a\ge c$

(v) $\Omega_{abc} \not\uparrow \{(k,x,y)\langle a,b,c\rangle \dashrightarrow (*,y,x)\langle a,c,b\rangle\}$

(vi) $\Omega_{abc} \not\uparrow \{(k,x,v)\langle a,b,c\rangle \dashrightarrow (y,*,x)\langle c,a,b\rangle\}$

Proof:

Let $a'=\log(a)$, $b'=\log(b)$, $c'=\log(c)$, and $r'=\log(r)$

(i) $\Omega_a \uparrow \{(k) \dashrightarrow (*)\}$, a 1-to-many broadcasting function, and $\Omega_{bc} \uparrow \{(x,y)\langle b,c\rangle \dashrightarrow (x,y)\langle b,c\rangle\}$, an identity.

Therefore, (i) is proved by applying Theorem 2.1.

(ii) First we will prove: $\Omega_{ab} \uparrow \{(k,x)\langle a,b\rangle \dashrightarrow (x,*)\langle b,a\rangle\}$.

Assume it is false, it implies that there exist x_i, x_j, r, p, q

such that $x_i \neq x_j$, $kb+x_i \underset{r}{\equiv} kb+x_j$ and $x_i a+p \overset{r}{\underset{ab}{\equiv}} x_j a+q$.

If $r \geq b$, we have $x_i \underset{b}{\not\equiv} x_j$ and $x_i \underset{r}{\equiv} x_j$, a contradiction.

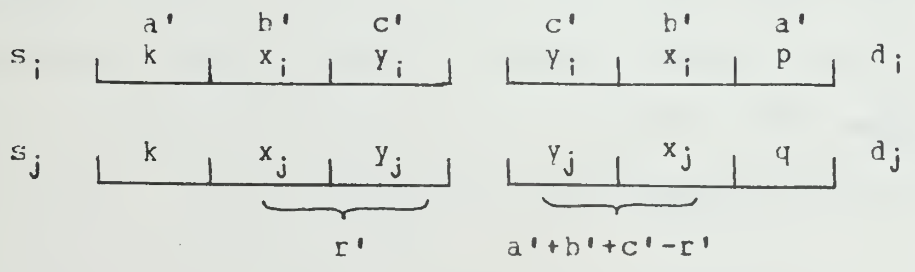
If $r<b$, the most significant $(a'+b'-r')$ bits of x_i will be the same as that of x_j, while the least significant r' bits of x_i will be the same as that of x_j. Since x_i and x_j have only b' bits each, then $x_i \underset{b}{\equiv} x_j$, which implies a contradiction.

Hence $\Omega_{ab} \uparrow \{(k,x) \dashrightarrow (x,*)\}$.

Using Theorem 2.1 again, noting that $\Omega_c \uparrow \{(y) \dashrightarrow (y)\}$, (ii) is proved.

(iii) Proof of (iii) is similar to that of $\Omega_{ab} \uparrow \{(k,x) \dashrightarrow (x,*)\}$ in (ii).

(iv) For this case, we can represent the tags (s_i, d_i) and (s_j, d_j) as follows:

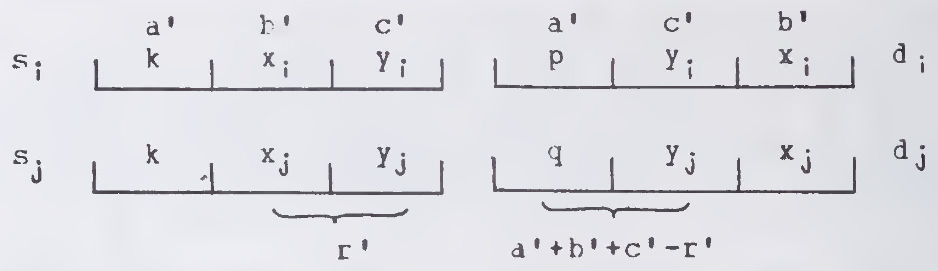


Each tag is divided into 3 parts of lengths a', b' and c' respectively.

If $a \geq c$, assume $\Omega_{abc} \not\uparrow \{(k,x,y)\langle a,b,c\rangle \dashrightarrow (y,x,*)\langle c,b,a\rangle\}$.

This implies that there exist s_i, d_i, s_j, d_j, and r such that $s_i \underset{abc}{\not\equiv} s_j$, and the least significant r' bits of s_i

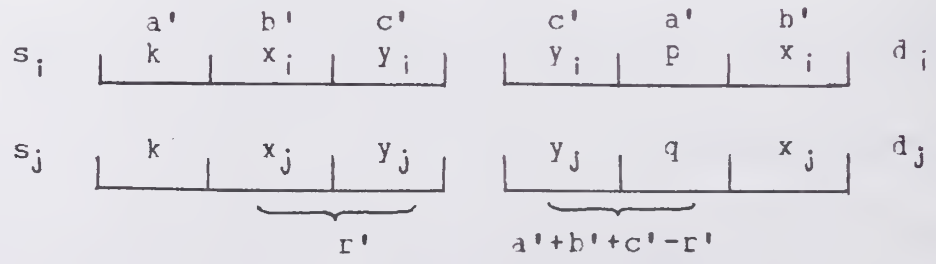
(v) This case can be represented pictorially as:



If $b \geq c$, we can pick (s_i, d_i) & (s_j, d_j) such that $p=q$, $y_i = y_j$, and the most significant $(b'-c')$ bits of x_i & x_j equal, with the least significant c' bits unique. Then $s_i \underset{abc}{\not\equiv} s_j$. By letting $r=c$, we can see that $s_i \underset{r}{\equiv} s_j$. Also we have the most significant $[a'+c'+(b'-c')]$ bits of d_i & d_j being equal, which is equal to $a'+b'+c'-r'$. Hence we just show that $\Omega_{abc} \updownarrow \{(k,x,y)<a,b,c> \dashrightarrow (*,y,x)<a,c,b>\}$.

If $c>b$, we pick (s_i, d_i) & (s_j, d_j) such that $p=q$, $y_i = y_j$ and $x_i \neq y_j$. Then $s_i \underset{abc}{\not\equiv} s_j$. We also pick $r=c$, then $s_i \underset{r}{\equiv} s_j$. The number of leading bits of d_i & d_j that are the same $=a'+c' > a'+b' = a'+b'+(c'-r')$. Hence we can see that $\Omega_{abc} \uparrow \{(k,x,y)<a,b,c> \dashrightarrow (*,y,x)<a,c,b>\}$.

(vi) We can represent this case as:



If $a'+b'+c' > 2c'+a'$ (i.e. $b'>c'$), we pick $y_i = y_j$, $p=q$ and the and s_j are equal, while the most significant (a'+b'+c'-r') bits of d_i and d_j are equal.

If $r \geq c$, then from s_i and s_j, we can see that $y_i = y_j$. Also, the least significant (r'-c') of x_i and x_j are equal. From d_i and d_j, we can see that the most significant (a'+b'+c'-r'-c') bits of x_i and x_j are equal. These add up to a'+b'-c' bits of x_i and x_j, and is greater than or equal to b' if $a \geq c$. So $x_i = x_j$. This contradicts $s_i \underset{abc}{\not\equiv} s_j$.

If $r<c$, we have the most significant (a'+b'+c'-r') bits of y_i and y_j equal, which is greater than a'+b' and thus greater than c'. So $y_i = y_j$. From an argument similar to that in the paragraph above, we have $x_i = x_j$. This also contradicts $s_i \underset{abc}{\not\equiv} s_j$.

Hence if $a \geq c$, $\Omega_{abc} \uparrow \{(k,x,y)<a,b,c> \dashrightarrow (y,x,*)<c,b,a>\}$.

If $a<c$, assume $\Omega_{abc} \uparrow \{(k,x,y)<a,b,c> \dashrightarrow (y,x,*)<c,b,a>\}$.

If $ab \geq c$, we pick (s_i,d_i) and (s_j,d_j) such that $y_i = y_j$ and $p=q$. Then $x_i = x_j$ except for the most significant (c'-a') bits. Then for r'=a'+b', we have $s_i \underset{r}{\equiv} s_j$ and $d_i \overset{r}{\underset{abc}{\equiv}} d_j$, and since $s_i \underset{abc}{\not\equiv} s_j$, we show that $\Omega_{abc} \not\uparrow \{(k,x,y)<a,b,c> \dashrightarrow (y,x,*)<c,b,a>\}$.

If $ab<c$, we simply have to pick $x_i = x_j$ and $y_i = y_j$ to arrive at the contradiction. So now, we get $\Omega_{abc} \uparrow \{(k,x,y)<a,b,c> \dashrightarrow (y,x,*)<c,b,a>\}$.

last (b'-c') bits of x & x equal. Then there will be conflict if r'=b'. This implies $\Omega_{abc} \not\uparrow \{(k,x,y)<a,b,c> \dashrightarrow (y,*,x)<c,a,b>\}$.

If $a'+b'+c' \le 2c'+a'$ (i.e. $b' \le c'$), we simply pick $y_i = y_j$, p=q and $x_i = x_j$. Then for r<c, we will have conflict, which implies $\Omega_{abc} \not\uparrow \{(k,x,y)<a,b,c> \dashrightarrow (y,*,x)<c,a,b>\}$.

These broadcasting theorems are also essential in establishing the recurrence algorithms in Chapter 3 and are some of the more important properties of the omega network.

2.2.4 General Admissibility

It is well known that the omega network can only pass a small fraction of the total number of N permutations(N**(N/2)/N!). To improve the permutation ability and to understand better the relationship between permutations and the omega and inverse omega networks, we would like to extend some of the results of Pease[16].

Without relabelling, the indirect binary n-cube array described in [16] is actually an inverse omega network[4] after rearrangement of the switches. We can extend his results to the omega network in a very simple manner. Let the index p be represented as $(p_1 2^{n-1} + p_2 2^{n-2} + \ldots p_{n-1} 2 + p_n)$ † instead of $(p_1 + 2p_2 + \ldots 2^{n-1} p_n)$, as in Formula 1 of [16]. Then we can use exactly the same theorems as in [16], except now we should note that the index bits are reversed.

Let x and y be expanded in binary notation as $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_n)$ with x_1 and y_1 being the most significant bits, x_n and y_n the least significant. The function describing a permutation P can be written as a set of functions

$$y_i = P_i(x_1, x_2, \ldots, x_n). \qquad (2.14)$$

† This notation is consistent with other sections of this thesis.

The principal theorem for the omega network will then be:

Theorem 2.4

P is admissible by an omega network if and only if the functions (2.14) defining P can be written in the form

$$y_i = x_i \oplus f_i(y_1, \ldots, y_{i-1}, x_{i+1}, \ldots, x_n) \qquad (2.15)$$

for $1 \leq i \leq n$. $\oplus$ is the 'exclusive or' operation.

It is a reformulation of Theorem 2 in [4]. However, it applies only to permutations. No broadcasting function is considered.

A similar theorem for inverse omega network will be:

Theorem 2.5

P is admissible by an inverse omega network if and only if the functions (2.14) defining P can be written in the form

$$y_i = x_i \oplus f_i(y_n, \ldots, y_{i+1}, x_{i-1}, \ldots, x_1) \qquad (2.16)$$

for $1 \leq i \leq n$. $\oplus$ is the 'exclusive or' operation.

Using the new rotation, we will redefine lower and upper triangular permutations as follows:

Definition 2.2

A permutation is lower triangular if the functions(2.15) defining P can be written as

$$y_1 = x_1 \oplus c$$
$$y_i = x_i \oplus f_i(y_1, y_2, \ldots, y_{i-1}), \qquad (2.17)$$

where $2 \le i \le n$, c=0 or 1.

Definition 2.3

A permutation is upper triangular if the functions(2.15) defining P can be written as

$$y_i = x_i \oplus f_i(x_{i+1}, x_{i+2}, \ldots, x_n)$$
$$y_n = x_n + c \qquad (2.18)$$

where $1 \le i \le n-1$, c=0 or 1.

All lower and upper triangular permutations are passable by omega network. We will now present the following two lemmas that show that they are also inverse omega passable.

Lemma 2.3

A lower triangular permutation can also be represented in the form $y_i = x_i \oplus g_i(x_1, \ldots x_{i-1})$, and is thus inverse omega passable.

Proof:

Assume $y_i = x_i \oplus g_i(x_1, \ldots, x_{i-1})$ for i=1,...,k.

$$\begin{aligned} y_{k+1} &= x_{k+1} \oplus f_{k+1}(y_1, y_2, \ldots, y_k) \\ &= x_{k+1} \oplus f_{k+1}(x_1 \oplus c, x_2 \oplus g_2(x_1), \ldots, x_k \oplus g_k(x_1 \ldots x_{k-1})) \\ &= x_{k+1} \oplus g_{k+1}(x_1, x_2, \ldots, x_k) \end{aligned}$$

Since it is true for $k=1$, this Lemma is thus proved by induction.

Lemma 2.4

An upper triangular permutation can also be represented in the form $y_i = x_i \oplus g_i(y_n, \ldots y_{i+1})$, and is thus inverse omega passable.

Proof:

The proof is similar to that of Lemma 2.3.

The following two theorems are taken out of [16], and will be presented without proofs.

Theorem 2.6 [Pease]

The set of admissible lower triangular permutations is a group under composition of maps.

Theorem 2.7 [Pease]

The set of admissible upper triangular permutations is a group under composition of maps.

Pease showed that shifts are lower triangular permutation in his context, which is upper triangular in our notations. We are going to show here that the odd-ordered vector unscrambling permutation is also upper triangular.

We first let the binary representation of the odd order, p, be $(p_1 p_2 \cdots\cdots p_{n-1} 1)$. If the source index is x, then the destination index will be $y=p*x$. The values of the y_i's are then

$$y_n = x_n$$
$$y_{n-1} = x_{n-1} \oplus p_{n-1} x_n \oplus s_{n-1}(x_n)$$
$$y_{n-2} = x_{n-1} \oplus p_{n-2} x_n \oplus p_{n-1} x_{n-1} \oplus s_{n-2}(x_{n-1}, x_n)$$
$$\cdot$$
$$\cdot$$
$$\cdot$$
$$y_i = x_i \oplus \sum_{j=i}^{n-1} p_j x_{n-j+1} \oplus s_i(x_{i+1}, x_{i+2} \cdots x_n)$$
$$= x_i \oplus f_i(x_{i+1}, \ldots\ldots, x_n)$$

where s_i is the carry from the lower ordered bits to the ith bit.

It is immediately obvious that this is an upper triangular permutation defined in Definition 2.3. Hence by Theorem 2.7, we can see that all compositions of shifts and odd-ordered unscrambling permutations are omega passable.

Defining $x=(x_1\ x_2 \ldots x_n)^t$, and $y=(y_1\ y_2 \ldots y_n)^t$. A permutation is linear if there exists an n x n nonsingular binary matrix, P, such that

$$y = P \cdot x \qquad (2.19)$$

Extending Pease' result to our notations, a linear permutation is omega passable if P can be decomposed into the matrix product LU, where L is a lower unit triangular matrix and U is an upper unit triangular matrix. (Unit means that all coefficients on the main diagonal are ones). By analogy, a linear permutation is inverse omega passable if P can be decomposed into UL.

One important result from [16] is that any nonsingular P can be decomposed into L_1UL_2. This implies that P can be decomposed as L_1U and L_2. So P can be passed by two omega passes. It can also be decomposed as L_1 and UL_2. Hence it can be passed by two inverse omega passes. This is a very significant result because it increases the permutation ability of the omega network. The perfect shuffle permutation is a good example. Although the omega network is made up of stages of perfect shuffles, a perfect shuffle permutation cannot be passed by a fixed log N stages omega network. However, it is a linear permutation. So it can be passed by two omega passes.

A diagram showing the permutation abilities of the omega and inverse omega networks is shown in Figure 2.8.

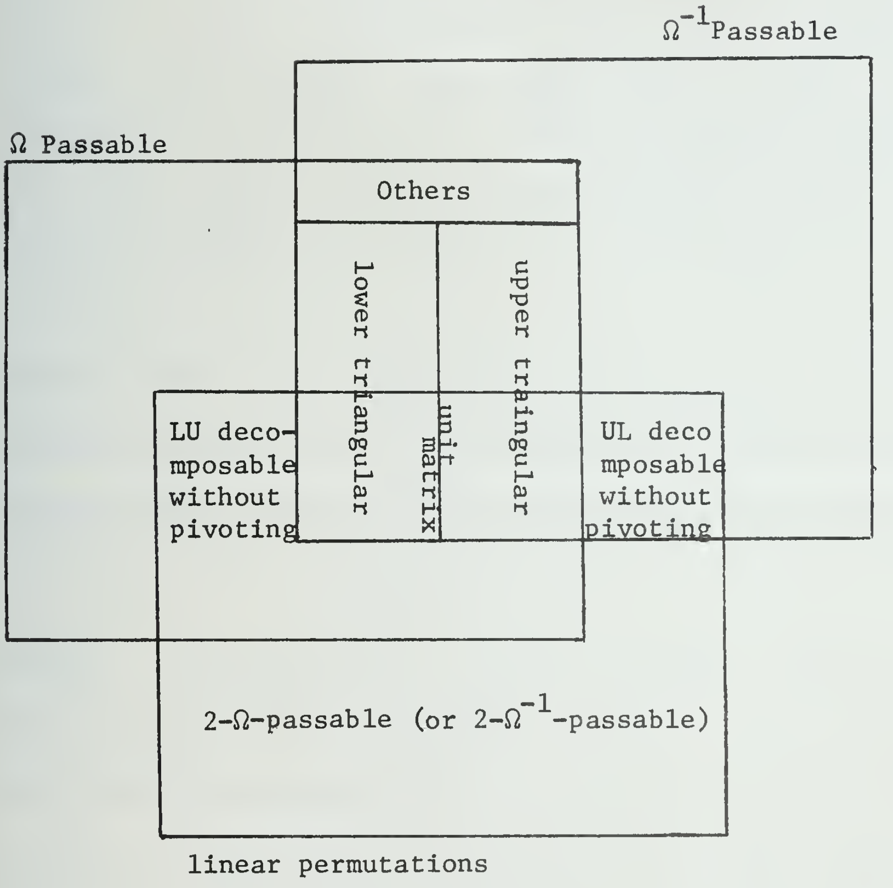

Ω^{-1}Passable
Ω Passable
Others
lower triangular
upper triangular
unit matrix
LU decomposable without pivoting
UL decomposable without pivoting
2-Ω-passable (or 2-Ω^{-1}-passable)
linear permutations

Figure 2.8 Permutation Abilities of Omega Network

and Inverse Omega Network

2.3 Batcher Network and the Shuffle Connection

One network that bears a great resemblence of the omega network, is the Batcher's merging network. The only structural difference is that instead of using tag bit comparison at each of the switching elements, the Batcher merger compares the magnitude of the two whole tag words at each switch to determine which of the two output ports to select.

Before we continue the discussion, we first define the order set of a set of N elements as the relative ordering of the elements. For example the order set of (8,12,17,3,9) is (1,3,4,0,2).

It can be observed that the Batcher's merging algorithm for a set of distinct elements is equivalent to the omega tag routing algorithm for the corresponding order set.

It should, then, be obvious that the omega partition theorems also applies to the Batcher merger.

Now we can deduce an alternate proof of Stone's implementation [8] of the bitonic sorter on the perfect shuffle network.

The basic idea of an NxN Batcher bitonic sorter(N being a power of 2) is quite simple. Given two sorted sequences of length L each, if the first sequence is in ascending order while the second sequence is in descending order, then the 2Lx2L Batcher merger will sort the bitonic sequence formed by the juxtaposition of the two sequences. A bitonic sorter consists of log N stages of merging networks, where stage i $(1 \leq i \leq \log_2 N)$ consists of $N/2^i$ bitonic mergers of size $2^i \times 2^i$. (Some switching elements will need to have their outputs reversed in order to produce the required descending order.) By the extension of the omega partition theorem we can use an NxN Bitonic merger to 'simulate' $N/2^i$ bitonic mergers of size $2^i \times 2^i$, by setting the switches in the first $(\log_2 N-i)$ columns straight through. Hence, the sorting algorithm in [8] implements the Batcher bitonic sorter on a perfect shuffle network.

2.4 Benes Network and the Shuffle Connection

It can be proved that the binary Benes network of size NxN (where N is a power of 2) is equivalent to a cascade of an inverse omega network and an omega network , with the middle two columns of switching elements collapsed into one. The best known control time for the Benes network [17] requires on the order of $N x \log_2 N$ operations. However, the control time for an omega network is $O(\log_2 N)$, so for all omega passable or inverse omega passable permutations, the control time is only on the order of $\log_2 N$.

2.5 One Stage Perfect Shuffle Network

Since many connection networks are shuffle-based, if we build a one stage perfect shuffle network to interconnect all the processors, we can simulate any of these networks by cycling sufficient number of times through the network. Moreover, the complexity of the network will only be O(N), which will be a great deal better than that of other networks.

One obvious shortcoming of the omega network is its inability to pass some permutations. In this section, we are searching for the best strategies to pass any permutation through a one stage recyclic perfect shuffle network. By recycling a one stage perfect shuffle network a sufficient number of times, we hope to be able to pass any general permutation.

Lang [18] proposed to use queues at the outputs of the switching elements, and then cycle the network for as many times as it is needed for each of the $\log_2 N$ steps. Following this strategy, the number of shuffle cycles required in the worst case is found to be

$=2\sqrt{2N}-3$ for $\log_2 N$ being odd,

$=3\sqrt{N}-3$ for $\log_2 N$ being even.

The length of the queues can grow to:

$\sqrt{N/2}$ for $\log_2 N$ being odd,

$\sqrt{N}$ for $\log_2 N$ being even.

Lang's algorithm is good in general. However, the building of two $O(\sqrt{N})$-long queues into each switching element certainly complicates the design of the switching elements. Hence in another possible strategy, we propose routing algorithms that do not require to use queues at the switches. The routing strategy is similar to that used in the Destination Tag Method. Every input port will generate a $\log_2 N$ bit tag representation of its destination and push it (together with the data) through the network. The switching functions of the switching elements are still the same. However, in case of conflict at any switch, one input will be honored while the other has to be switched to an undesired output port and restart from the most significant tag bit. At any given stage, the bit positions to be examined for each tag may be different. So we need a bit count associated with each tag to indicate which bit position will have to be examined. After the last tag bit of each input has been examined, it will be stored away in a register and taken off the network.

The conflict resolution can be:

a) gate straight.

b) honor the input furthest away from its destination.

c) honor the input nearest to the destination.

Simulation using random permutations shows that, on the average, all three resolutions are just about equally effective.

Instead of restarting from the most significant tag bit for the data at the wrong output port, we can use a built-in table to determine which bit to examine. In the discusson below, we let n equal to $\log_2 N$.

For the destination tag method, assume there is no conflict at any stage. We can observe that at stage k (k=2 to n), the data word whose destination tag is $d_1 d_2 \ldots d_n$ will be at switching element i (with binary representation $i_1 i_2 \ldots i_{n-1}$), where $d_1 d_2 \ldots d_k = i_{n-k} \ldots i_{n-1}$.

Hence by comparing the destination tags with the switch box numbers, we can find out how far a data word with certain destination tag is from its destination. A more useful information is which tag bit $d_k (1 \le k \le n)$ shall we examine at that particular switch for further switching. This number k is determined as the maximum number of trailing digits of the switch number i that match with the leading digits of the specified destination tag. These k values can be input as a built-in table in a ROM. For example, for an N=8 network, the table will be:

		Destination Tag (1st (log N-1) bits)			
		00	01	10	11
	00	3	2	1	1
Switch	01	1	3	2	2
Number	10	2	2	3	1
	11	1	1	2	3

Unfortunately, no theoretical bound has been found for any of these two startup methods. However, the two methods have been simulated for many random permutations and various network sizes. The simulated averages are tabulated below:

Network Size	8	16	32	64	128	256
Complete Restart	4.2	7.2	11.6	17.0	24.5	32.7
Table Lookup	3.9	6.5	10.0	14.7	20.2	27.9

Figure 2.9 compares these two restart methods with Lang's method. The table lookup method definitely has an advantage over the complete restart method, but is also slower than Lang's method.

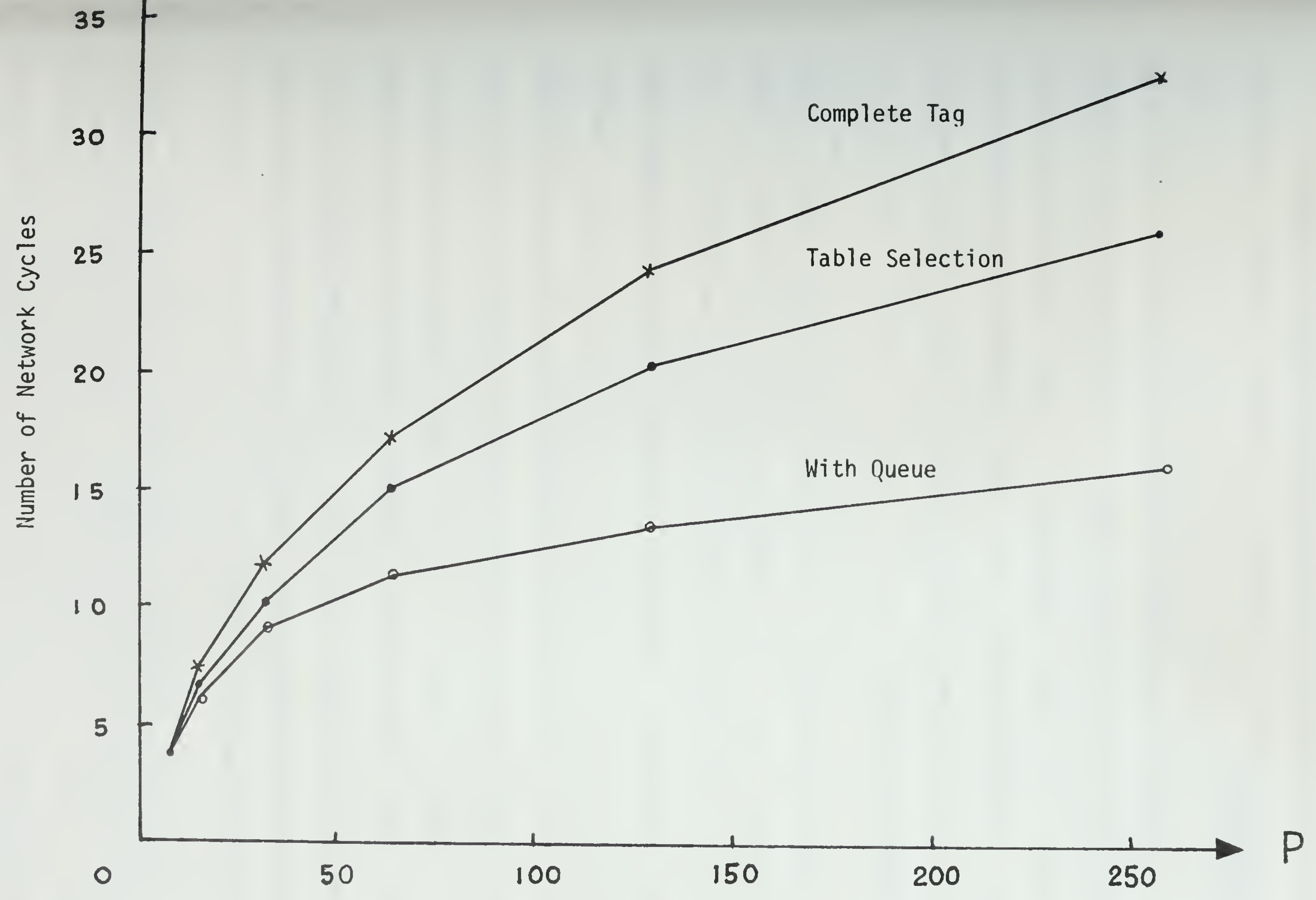


Figure 2.9 Simulated Means of Various Control Methods for 1-Stage Perfect Shuffle Network

3. NETWORK UTILIZATION IN PARALLEL PROCESSING SYSTEMS

3.1 Introduction

To build a meaningful processing system, we have to be able to handle efficiently most of the application demands of the users. In this section, we will investigate the alignment requirements of some common operations or algorithms and with what efficiency they can be handled by the alignment networks.

Array operations[†] are probably the most common type of operations found in ordinary Fortran programs and they have the most potential for high speedup and efficiency. So the most important criterion of a good parallel processing system is the efficient handling of array operations. Budnik and Kuck [19] and Lawrie [4] discussed ways of organizing the memories to allow conflict-free access to various slices of arrays. Linear skewing is a standard technique. However, the data output will sometimes form a p-ordered vector, which cannot be unscrambled by means of a simple shifter. [4] discussed the alignment requirements for some of the most common types of array accessings.

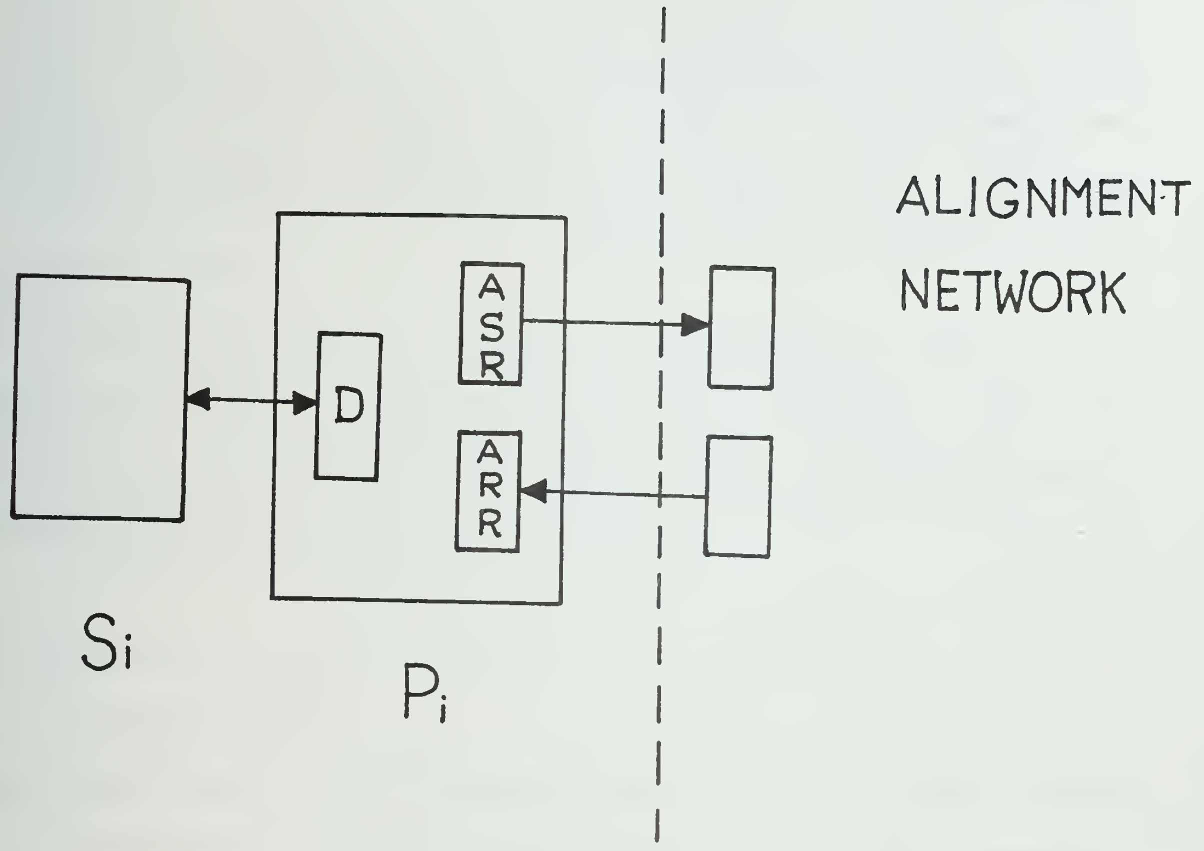
In ordinary programs, operations that are not scalar nor array operations very likely belong to the class of

[†] What we mean by array operations are the obvious type of vector operations found in programs, not the type we obtain by carefully rearranging the operation sequences of a particular algorithm.

recurrence operations. Recurrence operations, if not treated properly, will degrade a parallel processing system to a serial machine. Kogge and Stone [20], Heller [21], and Chen and Kuck [22] have shown various algorithms to speed up recurrence operations. Section 3.2 will discuss the adaptation of various recurrence solving algorithms onto parallel processing systems. It will be shown that, with careful planning, the alignment requirements can be greatly simplified. Hence, we would not need to use a full crossbar. Instead, a simpler alignment network, such as an omega network, will suffice.

The adaptation of recurrence operations onto parallel processing actually serves as a good example of how a well known computation algorithm can be tailored according to the limited number of available permutations of the alignent network to minimize alignment time. In the extreme cases, the alignment network may have only limited number of connections (like the Illiac IV shifter or a one-stage perfect shuffle network). To obtain any general permutation, the network has to be recycled many times. For example, the one-stage perfect shuffle network described in Section 2.5 may require $O(\sqrt{N})$ alignment steps before we can start on a processing step. By carefully rearranging some of the operation sequences in normal algorithms and by assigning intermediate storage patterns in a deliberate fashion, we can hopefully reduce the number of alignments per processing step down to a constant (not dependent on N). Pease [10] and Stone [8] showed how the Fast Fourier Transform can be done efficiently on a multiprocessing system interconnected with the perfect shuffle connection. In Section 3.3 we are going to show how matrix multiplication can be done in a more efficient way in a multiprocessing system with a certain class of connection networks. The number of alignment steps is shown to be reduced by a factor of $\sqrt{N}$ or $\log_2 N$.

The algorithms described in this section are in such details that they can be easily microprogrammed into the respective parallel processing systems. The intermediate storage and alignment patterns are all clearly specified. Masks are needed occasionally to prohibit some processors from doing the prescribed operations at some steps. Throughout this section, we are considering parallel processing systems structured like that in Figure 3.1. The central control unit is not shown in the figure, but is actually the master unit of the array of processors. It sends the microinstructions to all the processors together with the masks. Each processor will address only its own memory. If the data words obtained need to be sent to different processors, they will be gated to the Alignment Send Register(ASR). After the required alignment is done, they will be returned to the Alignment Receive Register(ARR). With this architecture, we can align internal registers as well as memory registers.



)	Memory Data Register
ASR	Alignment Send Register
ARR	Alignment Receive Register
S_i	Memory Module i
P_i	Processor i

Figure 3.1 A Parallel Processor System Configuration

3.2 Adaptation of Recurrence Solvers

Chen and Kuck [22] provided many good algorithms to handle recurrence systems. To actually implement these algorithms on a parallel processing system, one would require some careful partitioning of the recurrence system, and a good, uniform way to allocate the initial and intermediate data so as to minimize the data routing time and the amount of intermediate storage space.

The solution of a $R<n,m>$ recurrence system is actually equivalent to the solution of a banded unit lower triangular matrix system with matrix size $n \times n$ and the number of nonzero bands $=m+1$. In general, we have to solve (3.1) for x to get the recurrence results,

$$A\ x = f \qquad (3.1)$$

where A is a lower triangular matrix with 1's on the main diagonal and m more nonzero subdiagonals.

[23] and [24] reorganized some of the recurrence algorithms into partitioned matrix notations to simplify understanding. According to the number of processors available and the values of m and n, we have to use different recurrence solving algorithms for higher efficiency. In general, there are three major algorithms to handle recurrence systems. The first algorithm uses a limited number of processors, and evolves from [23] and Algorithm 5

of [25]. The second algorithm assumes the presence of a large number of processors, but will do the folding when the number of available processors is less than the upper bound. It evolves from [24] and Algorithm 2 of [22]. The third algorithm is similar to the second algorithm except it uses a less parallel method in solving the small full recurrence systems in the initialization stage. The number of processors used will be between that of the first and the second algorithms.

Given p(the number of processors), and values of m and n, the execution time for the first algorithm

$=2mn(m+2)/p+\log_2(p/m)*(m^2+3m/2+1)-m^2-9m/2-2,$ for $p<n$,

$=2m(m+2)+\log_2(n/m)*(m^2+3m/2+1)-m^2-9m/2-2,$ for $p\geq n$.

The execution time for the second algorithm

$=2m\log(n/m)+4m,$ for $p=mn$,

$=(4m\log(n/m)+6m)mn/(2p),$ for $p<mn/2$,

$=4m\log(n/m)+6m,$ for $p>mn$.

The execution time for the third algorithm

$=\log_2 n(2+\log_2 m)-\log_2 m(\log_2 m+1)/2,$ for $p\geq m^2 n$,

$=(\log_2 n(2+\log_2 m)-\log_2 m(\log_2 m+1)/2)m^2 n/p,$ for $p<m^2 n$.

We can decide which algorithm to use by comparing the above times, given m, n, and p. A diagram showing what algorithm to use for m=8 is shown in Figure 3.2. We can see that if we have many processors, we would like to use the second algorithm; if we have a little bit less processors, the third algorithm will be more efficient; and if we only have a limited number of processors we would like to use the first algorithm. A similar pattern can be observed for m's of other values.

m=8:

	n sizes							
p	16	32	64	128	256	512	1024	2048
4	a	a	a	a	a	a	a	a
8	a	a	a	a	a	a	a	a
16	a	a	a	a	a	a	a	a
32	a	a	a	a	a	a	a	a
64	c	a	a	a	a	a	a	a
128	c	c	c	a	a	a	a	a
256	b	c	c	c	a	a	a	a
512	b	b	c	c	c	a	a	a
1024	b	b	b	c	c	c	a	a
2048	b	b	b	b	c	c	c	a
4096	b	b	b	b	b	c	c	c

a ------ choose the first algorithm.

b ------ choose the second algorithm.

c ------ choose the third algorithm.

Figure 3.2 Choice of Recurrence Algorithms for m=8

3.2.1 Using a Limited Number of Processors

The original algorithm can be found in [23]. The following algorithm shows how it can be adapted to a parallel processing system.

We will first describe how the input-output arrays and the intermediate arrays are stored in the parallel memory system. The nonzero elements of the A matrix, other than the main diagonal, are stored in a section of the memory called L. The processors and the memories are broken into k partitions (numbered 0 through (k-1)) of size m, where $k=p/m$. The jth off-diagonal element of the (i+1)th row of A will be stored in the (m-j)th memory of the $\lfloor ik/n \rfloor$th partition with relative address= i mod m. The f vector is stored in the memory section called f. The f vector is broken into k subvectors. The i-th element of the j-th subvector will be stored in the (i mod m)th memory of the jth partition with relative address=$\lfloor i/m \rfloor$. G and H are intermediate storage sections. The result vector x is stored in two memory sections, v and y. v and y together can actually be the alias of memory section f, and the x vector is stored in exactly the same way as the f vector. Section y is the alias for the first (n/p-1) locations of section f and v is the alias for the last location (relative address=n/p-1) of f.

For maximum efficiency of this algorithm, p should be less than or equal to n.

In the following algorithm,

PPN = processor identification number ($0 \leq PPN < p$),

RA = relative address within a memory section,

ACC = accumulators in the processors,

R1-R3 = the three internal registers in the processors.

<u>Algorithm</u>:

Stage 1:

```
1. Form k partitions of size m.‡
2. Set ppn = PPN mod m.
3. Repeat for i=1 to n/k-1;
  a. if ppn $\geq$(i mod m), then x=0.
                              else x=1.
  b. fetch from f, RA = $\lfloor i/m \rfloor$+x.
  c. left shift by (i mod m) into ACC.
  d. fetch from L, RA=m-ppn+i-1.
  e. perform a flip route in m-partition into R1.
  f. fetch f element from memory(i mod m), RA=$\lfloor i/m \rfloor$.
```

‡ The 'Form Partition' command forces all subsequent instructions to conform to this partitioning, for both alignment and memory accessing, unless specified otherwise.

```
  g. broadcast to respective m-partition into R2.
  h. multiply R1 and R2 into R3.
  i. subtract R3 from ACC.
  j. right shift ACC by (i mod m).
  k. store ACC into f, RA=$\lfloor i/m \rfloor$+x.
4. Done.
```

Stage 2:

```
1. Repeat for i=0 to m-1;
  a. fetch from I, RA =i, memories 0 through (m-i-1).
  b. right shift by i.
  c. store into G, RA=i.
2. Form k partitions of size m.
3. Repeat for i =1 to n/k-1;
  a. fetch from G, RA =i, into ACC.
  b. repeat for j =0 to m-1;
    (i)    set h = i+j-m.
    (ii)   if h<0 then set R1=0.
                 else fetch from G, RA=h, into R1.
    (iii)  fetch L element from memory j, RA=i-1.
    (iv)   broadcast to respective m-partitions into R2.
    (v)    multiply R1 and R2 into R3.
    (vi)   subtract R3 from ACC.
  c. store ACC into G, RA=i.
4. Done.
```

Stage 3:

```
A.  Perform an m x m matrix transpose in m-partition from
    the last m rows of G's to H.
B.  Fetch from f, RA=(n/p-1), and store into v.
C.  Repeat for j=0 to log(p/2m);
  1. set r = (2**j)*m.
  2. Form (p/2r) partitions of size 2r. Subdivide each
     partition into left and right halves. $v^{(j)}_{2i-1}$ and $H^{(j)}_{2i-2}$
     in the left, and $v^{(j)}_{2i}$ and $H^{(j)}_{2i-2}$ will be in the right,
     $1 \leq i \leq (p/2r)$.
     ($H^{(j)}_{o}$ is nonexistent and filled with all 0's)
  3. Fetch from the right half v into right half ACC.
  4. Put 0's into left half ACC.
  5. Repeat for h=0 to m-2 by 2;
    a) fetch from right half H, RA=h.
    b) left shift by r into left half R1.
    c) fetch from right half H, RA=h+1, into right half R1.
    d) fetch from left half v, memories(r-m+h) and (r-m+h+1)
       only.
    e) broadcast respectively to left and right half R2.
    f) multiply R1 and R2 into R3.
    g) subtract R3 from ACC.
  6. Right shift ACC in left half by r into R1 of right
     half.
  7. Add right half R1 to right half ACC.
  8. Store right half ACC into right half v.
```

```
 9. If j=log(p/2m) go to D.
10. Repeat for h =0 to m-2 by 2;
   a) set ACC=0.
   b) repeat for q=0 to m-1;
      (i)      fetch from right half H, RA=q.
      (ii)     fetch H element from left half memory
               (r-m+q), RA=h.
      (iv)     broadcast into left R2.
      (v)      fetch H element from left half memory
               (r-m+q), RA=h+1.
      (vi)     broadcast into right R2.
      (vii)    multiply R1 and R2 into R3.
      (viii)   add R3 to ACC.
   c) negate ACC.
   d) store left ACC into right half H, RA=h.
   e) store right ACC into right half H, RA=h+1.
D. Done.
```

Stage 4:

```
1.  Form k m-partitions.
2.  Repeat for h=0 to n/p-2;
  a. Perform an m x m matrix transpose from G(RA:(hm) to
     m(h+1)-1) to H.
  b. Fetch from f, RA=h, into ACC.
  c. Repeat for i=0 to m-1;
```

```
        (i)    fetch from H, RA=i, into R1.
        (ii)   fetch v element from memory i.
        (iii)  broadcast to the right neighbor m-partition into
               R2.
        (iv)   multiply R1 and R2 into R3.
        (v)    subtract R3 from ACC.
    d. Store ACC into y, RA=h.
3.  Done.
```

<u>Analysis</u>:

Throughout the algorithm, partitions of size m or 2r (=2^{j+1}m) are used. By Corollary 2.1, we can see that the omega network (or some of the full permutation networks) is capable of performing the necessary alignments because of its partition ability. As for the different kinds of alignment patterns that are required within the partitions, we have right and left shifts, flips and 1-to-many broadcasting. All of these patterns can be passed by the omega network. One of the noteworthy patterns can be found in step C.5.e of Stage 3. The broadcasting function has the form $\{(k,x)\langle r,2\rangle \text{---}\!\!> (x,*)\langle 2,r\rangle\}$. That this function can be passed by the omega network is proved in part (ii) of Section 2.2.1.2 of this thesis. In step 2.c.(iii) of Stage 4, the connection function can be passed by the omega network, by virtue of Theorem 2.1, after setting P_L to a 1-shift permutation and

P_M's to 1-to-many broadcasting.

The m x m matrix transpose in step A of Stage 3 and step 2.a of Stage 4 can be implemented as a 'subroutine' that takes (m-1) steps.

Assume element (i,j) of matrix M is stored in memory j with relative address i ($0 \leq i,j < m$).

Let PPN be the processor identification number and $\oplus$ be the bit by bit 'exclusive or' of two integers.

<u>Matrix Transpose Routine</u>:

```
 DO k = 1 to m-1;
        h = PPN ⊕ k
        Fetch element with index h into R1
        Swap R1 with processor h*
        Store R1 into M, RA=h
 END
```

The detailed counts for various operation times in this algorithm are listed below:

Stage 1:

Fetch:	3(n/k-1)
Store:	(n/k-1)

* For omega networks, we can use the column control method described in Section 2.1.4, with p set to h.

Alignment:	$4(n/k-1)$
Processor:	$2(n/k-1)$

Stage 2:

Fetch:	$2mn/k+n/k-m-1$
Store:	$n/k+m-1$
Alignment:	mn/k
Processor:	$2mn/k-2m$

Stage 3:

Fetch:	$\log_2(p/m)(3m^2/2+3m/2+1)-3m^2/2+m+1$
Store:	$\log_2(p/m)(m+1)$
Alignment:	$\log_2(p/m)(3m^2/2+2m+2)-3m^2/2$
Processor:	$\log_2(p/m)(m^2+3m/2+1)-m/2-m^2$

Stage 4

Fetch:	$3mn/p-3m+n/p-1$
Store:	$(n/p-1)(m+1)$
Alignment:	$2mn/p-2m$
Processor:	$2mn/p-2m$

As for the total, we get:

Fetch:	$O(3m^2\log_2(p/m)/2+2m^2n/p)$
Store:	$O(m.\log_2(p/m)+3mn/p)$
Alignment:	$O(3m^2\log_2(p/m)/2+m^2n/p)$
Processor:	$O(m^2\log_2(p/m)+2m^2n/p)$

As we can see from these figures, the alignment and memory times are of the same order as the processor time. This implies that we are not spending excessive delay in either the accessing or alignment of the intermediate data elements.

3.2.2 Full Recurrence System Solver

(using $p=n^3$ processors)

The algorithm we use here is derived from [23]. The essence of the algorithm is as follows:

Assume we have to solve for x in

$$L\,x = f \tag{3.2}$$

where L is a unit lower triangular matrix of size n x n, while x and f are arrays of sizes n x h. The inverse of L can be represented as,

$$L^{-1} = M_{n-1}M_{n-2}\cdots\cdots M_1. \tag{3.3}$$

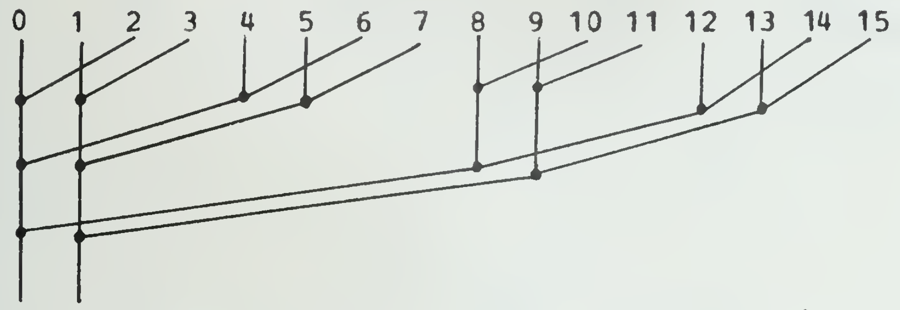
where $M_i=(I-L_ie_i^t)$ in which L_i is the ith column of L with the element L(i,i) set to zero. The solution x is then given by

$$x = M_{n-1}M_{n-2}\cdots\cdots M_1f. \tag{3.4}$$

Then we will solve this product in parallel using $\log_2 n$ stages.

The initial storage patterns for the arrays in the n^3 processors/ memories are shown in Figure 3.3.

For example, if $2n^2 = 2$, then the addition tree will look like:



For the $M^{(j+1)}$ calculation, a pictoral description of what needs to be done is shown in Figure 3.4.

Here, similar to what we do in the calculation of $G^{(j+1)}$, we broadcast $M^{(j)}_{2i+1}$ elements to the right side R1's and $T^{(j)}_{2i}$ elements to the right side R2's. The partial results will then be in $R3(1,0,i,*,x,y)<2,n/\bar{r},n/2r,r,\bar{r},n>$. The summation of partial products are done by shifts (of $2^h.r.n$) and add, $0 \leq h<j$. The multiplication and additions will be done at the same time as those in the calculation of $G^{(j+1)}$.

We first define S as the memory system of n^3 modules and P as the processor system of n^3 modules. In this algorithm, we need four internal processor registers (R1 through R4).

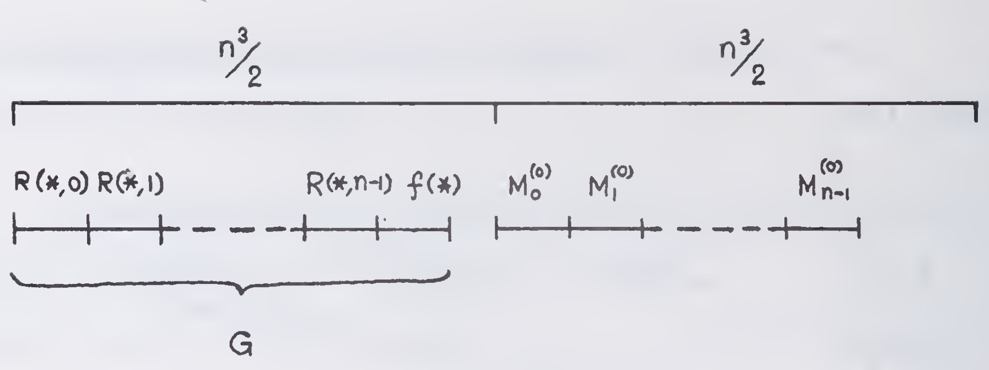


Figure 3.3 Initial Array Storage

G consists of the h right hand columns and $M_i^{(0)}$ is the ith column of the left hand matrix, L, $(1 \le i \le n-1)$. Note that $M_0^{(0)}$ is all zeros and the nth column of the matrix has no entry.

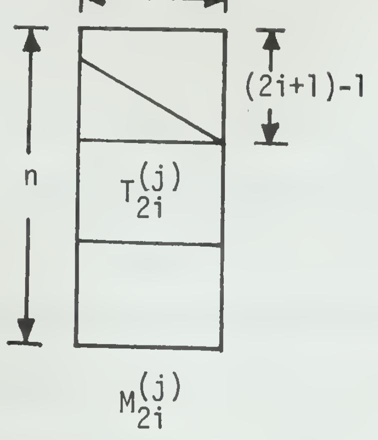
Before we proceed, we would like to introduce some new notations to simplify the description of the algorithms.

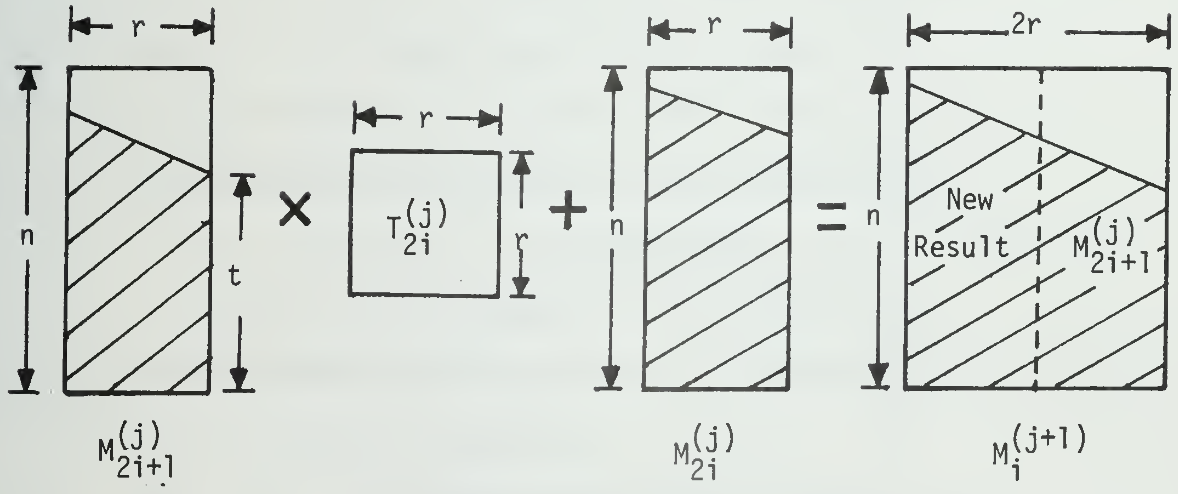
#N ---- the set {0,1,2,....N-1}

* ---- the set {.....,-2,-1,0,1,2,.....}

All vectors are declared using the notation A<U> and are indexed from A(0) to A(U-1).

For calculation of $G^{(j+1)}$, we will first broadcast M1 to the left half R1's. Then we broadcast Y elements to the left half R2's. Then the partial results of $G^{(j+1)}$ will be in R3(0,0,*,x,y)<2,n/4r,r,2n,n>. The summation of partial results are done by shifts (of $2^{h+1} * n^2$) and add, $0 \le h < j$.





$t=n+1-(i+1)2r$

$i=1,2,\ldots,(n/2r-1)$

Figure 3.4 $M_i^{(j+1)}$ Calculation

The algorithm is shown below:

<u>Algorithm</u>:

```
A. Repeat for $j = 0$ to $(\log_2 n-2)$;
   1. Let r=2**j, r'=max(r/4,1), r"=max(r/2,1), $\bar{r}$=2r".
   2. Declare S(RA=G) as Q<2,n/4,2n,n>.
      Declare S(RA=M) as W1<2,n/2r',n/r,r',r,n>.
      Declare S(RA=M) as W2<2,n/2r",n/2r,r",2r,n>.
      Then for i=0 to (n/2r-1),
        $G^{(j)}$ <2n,n> is in Q(0,0,*,*)
        $Y^{(j)}$ <2n,r> is in Q(0,0,*,#r+(r-1))
        $M^{(j)}_{2i}$ <r,n> is in W1(1,0,2i,0,*,*), $M^{(j)}_{o}$ is
              immaterial.
        $M^{(j)}_{2i+1}$ <r,n> is in W1(1,0,2i+1,0,*,*),
        $T^{(j)}_{2i}$ <r,r> is in W1(1,0,2i,0,*,#r+(2i+1)r-1),
        $M^{(j+1)}_{i}$ <2r,n> is in W2(1,0,i,0,*,*).
   3. Declare P as P1<2,n/4r,r,2n,n>.
      Declare P also as P2<2,n/$\bar{r}$,n/2r,r,$\bar{r}$,n>.
      Then $G^{(j+1)}$ calculation uses P1(0,0,*,*,*),
      while $M^{(j+1)}_{i}$ calculation uses P2(1,0,i,*,*,*).
   4. Fetch $M^{(j)}_{1}$ (*,*) from W1(1,0,1,0,*,*).
   5. Broadcast D(1,0,1,0,x,y)‡ to R1(0,0,x,*,y) of P1,
      ∀x,y.
```

‡ D is memory data register. Declaration always follows whatever is to be fetched or stored.

```
6.  Fetch $M^{(j)}_{2i+1}$ (*,*) from W1(1,0,2i+1,0,*,*).
7.  Broadcast D(1,0,2i+1,0,x,y) to R1(1,0,i,x,*,y) of P2,
    $\forall$x,y.
8.  Fetch $Y^{(j)}$ (*,*) from Q(0,0,*,#r+(r-1)).
9.  Broadcast D(0,0,x,y) to R2(0,0,y,x,*) of P1, $\forall$x, and
    $\forall$y$\in${#r+(r-1)}.
10. Fetch $T^{(j)}_{2i}$ (*,*) from W1(1,0,2i,0,*,#r+(2i+1)r-1).
11. Broadcast D(1,0,2i,0,x,y) to R2(1,0,i,y,x,*) of P2,
    $\forall$x, and $\forall$y$\in${#r+(r-1)}.
12. Multiply R1 and R2 into R3.
13. Repeat for q=0 to (j-1);*
    a) Set R4=0.
    b) Declare P as P3<2,$n^2$/(2**(q+2)$\bar{r}$),2,$2^q$,$\bar{r}$,n>.
    c) Left shift R3(1,*,1,0,*,*) of P3 by $2^q\bar{r}$n into R4.
    d) Declare P as P4<2,n/2**(q+3),2,$2^q$,2n,n>.
    e) Left shift R3(0,*,1,0,*,*) of P4 by $2^q$.$2n^2$ into R4.
    f) Add R3 and R4 into R3.
14. Fetch $M^{(j)}_{2i}$ (*,*) from W1(1,0,2i,0,*,*).
15. Right shift D(1,0,2i,0,x,y) by (ir n/2) to R2
    (1,0,i,x,y),$\forall$x,y.
16. Fetch $G^{(j)}$ (*,*) from Q(0,0,*,*).
17. Transfer** D(0,0,x,y) to R2(0,0,0,x,y) of P1, $\forall$x,y.
```

---

* This step will be skipped when j=0.

** Transfers need no alignment.

```
    18. Add R2 and R3 into R2.
    19. Transfer R2(0,0,0,x,y) of P1 to D(0,0,x,y) of Q.
    20. Store D(0,0,*,*) to $G^{(j+1)}$ (*,*).
    21. Fetch $M_{2i+1}^{(j)}$ (*,*) from W1(1,0,2i+1,0,*,*).
    22. Right shift D(1,0,2i+1,0,x,y) of W1 to D
        (1,0,i,0,x+r,y) of W2 by $(ir^2n/2-r^2n/4+rn)$ $\forall$ x,y.
    23. Transfer R2(1,0,i,0,x,y) of P2 to D(1,0,i,0,x,y) of
        W2, $\forall$ x,y.
    24. Store D(1,0,i,0,x,y) into $M_i^{(j+1)}$ (x,y) $\forall$x,y.

B. For j = $\log_2 n-1$;
     1. Let r=n/2.
     2. Declare P as P5<n/2,2n,n>.
     3. Declare S(RA=G) as Q1<n/2,2n,n>.
        Declare S(RA=M) as W3<2,8,n/8,n/2,n>.
        Then  $G^{(j)}$ <2n,n/2> is in Q1(0,*,*),
              $Y^{(j)}$ <2n,n/2> is in Q1(0,*,#(n/2)+(n/2-1)),
              $M_1^{(j)}$ <n/2,n> is in W3(1,1,0,*,*).
     4. Fetch $M_1^{(j)}$ (*,*) from W3(1,1,0,*,*).
     5. Broadcast D(1,1,0,x,y) to R1(x,*,y) of P5, $\forall$x,y.
     6. Fetch $Y^{(j)}$ (*,*) from Q1(0,*,#(n/2)+(n/2-1)).
     7. Broadcast D(0,x,y) to R2(y,x,*) of P5, $\forall$x, and
        $\forall y \in \{$#(n/2)+(n/2-1)$\}$.
     8. Multiply R1 and R2 into R3.
     9. Repeat for q=0 to j-1;
        a) Set R4=0.
        b) Declare P as P6<n/2**(q+2),2,$2^q$,2n,n>.
```

c) Left shift R3(*,1,0,*,*) of P6 by 2^q.2n.n into R4.

d) Add R3 and R4 into R3.

10. Fetch $G^{(j)}$ (*,*) from Q1(0,*,*).

11. Transfer D(0,x,y) to R2(0,x,y) of P5, ∀ x,y.

12. Add R2 and R3 into R2.

13. Transfer R2(0,x,y) of P5 to D(0,x,y) of Q1, ∀ x,y.

14. Store D(0,*,*) to $G^{(j+1)}$ (*,*).

C. Done.

Analysis:

Steps A.5 and A.7 use a broadcasting function that is omega passable. We first apply part (ii) of the broadcasting theorems which shows that {(k,x,y)<2n,r,n>--->(x,*,y)<r,2n,n>} is omega passable. Then we can apply the omega partition theorem to allow for the shift in partitions. The broadcasting function in Step A.9 and A.11 are of the form {(k,x,y)<n,r,n>--->(y,x,*)<n,r,n>}. They are also omega passable because of Part (iv) of the broadcasting theorem (notice that a$\geq$c since a=c=n)), and the omega Partition Theorem. Step A.13 is the repetitive shifts and adds described earlier in this section. The broadcasting function in Step B.5 is of the form {(k,x,y)<2n,n/2,n>---> (x,*,y)<n/2,2n,n>} and that in Step B.7 is of the form {(k,x,y)<n,2n,n>---> (y,x,*)<n,2n,n>}. Both are passable by omega network.

The operation times for this algorithm are:

Fetch : $7\log_2 n-4$

Store : $2\log_2 n-1$

Align : $(\log_2 n) +4\log_2 n+1$

Processor: $\log_2 n(\log_2 n+3)/2$

3.2.3 Using Many Processors

This algorithm is derived from [24]. We will solve R<n,m> with $p=m^2n$. However, if the number of available processors is less than this, we will have to use folding.

The theoretical processor bound found in [22] is $m(m+1)n/2-m^3$. However, if m and n are powers of two, for p to be a power of two also, we have to use $p=m(2m)n/2=m^2n$.

The matrix L and the vector f can be written in the form,

$$L = \begin{pmatrix} L_0 & & & & \\ R_1 & L_1 & & & \\ & R_2 & L_2 & & \\ & & & \cdots & \\ & & & R_{\frac{n}{m}-1} & L_{\frac{n}{m}-1} \end{pmatrix}, \quad f = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{\frac{n}{m}-1} \end{pmatrix}$$

where L_i and R_i are m x m unit lower triangular and upper triangular matrices, respectively. Premultiplying both sides $Lx = f$ by the matrix $D = \left[\text{diag } L_i^{-1}\right]$, we obtain the system $L^{(c)} x = f^{(0)}$ where,

$$L^{(0)} = \begin{bmatrix} I_m & & & & \\ G_1^{(0)} & I_m & & & \\ & G_2^{(0)} & I_m & & \\ & & \cdot\ \cdot\ \cdot\ \cdot\ \cdot & & \\ & & & G_{\frac{n}{m}-1} & I_m \end{bmatrix}$$

and

$$L_0 f_0^{(0)} = f_0 \ ,$$

$$L_i \left[G_i^{(0)} \mid f_i^{(0)} \right] = \left[F_i \mid f_i \right] \quad i=1,2,\ldots n/m-1.$$

This will be called the initialization part of the algorithm, for it sets up the data for the main part of the algorithm.

Then for the main part, we form the sequence $L^{(j+1)}$, and $f^{(j+1)}$ for $j=0,1,\ldots,\log_2(n/m)-1$. Each matrix $L^{(j)}$ is of the form

$$L^{(j)} = \begin{bmatrix} I_r & & & & \\ G_1^{(j)} & I_r & & & \\ & G_2^{(j)} & I_r & & \\ & & & \cdots & \\ & & & G_{\frac{n}{r}-1} & I_r \end{bmatrix}$$

where $r = 2^j m$. For the (j+1)th stage, we have

$$G_i^{(j+1)} = \begin{bmatrix} 0 & G_{2i}^{(j)} \\ 0 & -G_{2i+1}^{(j)} G_{2i}^{(j)} \end{bmatrix}, \quad f_i^{(j+1)} = \begin{bmatrix} f_{2i}^{(j)} \\ -G_{2i+1}^{(j)} f_{2i}^{(j)} + f_{2i+1}^{(j)} \end{bmatrix}, \quad i = 0, 1, \ldots \frac{n}{2r}-1$$

The initialization part will be done using the method described in Section 3.2.2. We will not repeat the discusion here. The second stage of the algorithm, however, needs some discussion.

At step j+1 $(0 \le j < \log_2(n/m))$, we calculate $G_{2i+1}^{(j)} \cdot G_{2i}^{(j)}$ and $G_{2i+1}^{(j)} \cdot f_{2i}^{(j)}$ at the same time $(1 \le i \le (n/2r))$. Each calculation uses rm processors. The G calculation is done in the left $m^2n/2$ portion of the $p(=m^2n)$ processors while the f calculation is done in the right $m^2n/2$ portion. $G_{2i+1}^{(j)}$ has to be broadcasted to both portions.

The memory system of m^2n is broken into $r^2m/2$-partitions. $G_i^{(j)}$ $(0\leq i<n/r)$ is stored in $(0,i,0,*,*)$ $<2,n/r,m/2,r,m>$, while $f_i^{(j)}$ $(0\leq i<n/2r)$ is stored in $(1,i,0,*,0)$ $<2,n/r,m/2,r,m>$.

The f calculation will need one extra step to add $f_{2i+1}^{(j)}$ to the product $G_{2i+1}^{(j)}\cdot f_{2i}^{(j)}$.

<u>Algorithm</u>:

Stage 1 (Initialization) :

1. If m=1, the system is already initialized, we can go directly to Stage 2.
2. If m=2, there is only one off-diagonal element in each of the L_i's, $0\leq i<n/2$. We can solve each of the n/2 systems in $(0,i,*)<2,n/2,4>$ in two steps:

   $\bar{G}_i(1,*)=\bar{G}_i(1,*)-L_i\cdot\bar{G}_i(0,*)$, where $\bar{G}_i=[R_i|f_i]$.

   R_i will be in $(0,i,*,*)<2,n/2,2,2>$ and f_i at $(1,i,*,0)$ $<2,n/2,2,2>$ respectively.

   We then require two fetches and aligns to route them to $G_i^{(0)}$ at $(0,i,0,*,*)<2,n/2,1,2,2>$ and $f_i^{(0)}$ at $(1,i,0,*,0)$ $<2,n/2,1,2,2>$ respectively.
3. If m>2, we will use the method described in Section 3.2.2.

   The $\bar{G}_i$ calculation will be done in $(0,i,*)<2,n/m,m^3/2>$

and M calculation be done in (1,i,*)<2,n/m,m /2> for $0 \le i < n/m$.

Initially, R_i will be stored in column major order in (0,i,0,*,*)<2,n/m,m/2,m,m>. Resulting G_i and f_i will be in where R_i and f_i were.

For stage 2, we want $G_i^{(0)}$ to be in (0,i,0,*,*) <2,n/m,m/2,m,m> in row major fashion and $f_i^{(0)}$ to be in (1,i,0,*,0)<2,n/m,m/2,m,m>.

Hence we want to route (0,i,0,x,y) to (0,i,0,y,x), and also (0,i,1,0,z) to (1,i,0,z,0) $\forall$ x,y,z. Both routes are linear permutations and can be realized by the omega network in two passes, due to the results described in Section 2.2.4.

Stage 2 :

```
A. Repeat for j = 0 to log(n/2m);
     1. Let r=2**j.m.
     2. Declare S(FM=G or f) as M<2,n/2r,m,r,m>.
        Then for 0≤i<n/2r,
        $G_{2i}^{(j)}$ <r,m> is in M(0,i,0,*,*), $G_0^{(j)}$ being all 0's.
        $G_{2i+1}^{(j)}$ <r,m> is in M(0,i,m/2,*,*),
        $f_{2i}^{(j)}$ <r> is in M(1,i,0,*,0),
        $f_{2i+1}^{(j)}$ <r> is in M(1,i,m/2,*,0), rest of M(1,*,*,*,*)
        =0.
     3. Declare P as P1<2,n/2r,m,r,m>.
```

```
    4. Fetch $G^{(j)}_{2i+1}$ (*,*) from M(0,i,m/2,*,*).
    5. Broadcast D(0,i,m/2,x,y) to R1(*,i,y,x,*)
       of P1, $\forall$ x,y.
    6. Fetch $G^{(j)}_{2i}$ (#m+(r-m),*) and $f^{(j)}_{2i}$ (#m+(r-m)) from
       M(*,i,0,#m+(r-m),*).
    7. Broadcast D(z,i,0,x,y) to R2(z,i,x,*,y) of P1,
       $\forall$y,z, and $\forall$x$\in${#m+(r-m)}.
    8. Multiply R1 and R2 into R3.
    9. Repeat for q= 0 to ($\log_2 m$-1);
       a) Declare P as P2<mn/(2**(q+1)r),2,$2^q$,r,m>.
       b) Left shift R3(*,1,0,*,*) of P2 by 2 rm into R4.
       c) Add R3 and R4 into R3.
   10. Set R4=0.
   11. Fetch $f^{(j)}_{2i+1}$ (*) from M(1,i,m/2,*,0).
   12. Left shift D(1,i,m/2,*,0) into R4 by $rm^2/2$.
   13. Subtract R3 from R4 into R4.
   14. Right shift R4 by rm into D.
   15. Store D(*,i,1,*,*) into M(*,i,1,*,*).

B. Done.
```

<u>Analysis</u>:

The broadcasting function in Step A.5 of Stage 2 is of the form {(k,x,y)<m,r,m>--->(y,x,*)<m,r,m>} and is omega passable. The broadcasting function in Step A.7 of Stage 2 is of the form $\{(k,x,y)<m,r,m> \text{---}> (x,*,y)<m,r,m>\}$ and is also omega passable.

The operation times for Stage 1 can be easily obtained by substituting n by m in the operation times listed in Section 3.2.2. However, we have to add four alignment passes for the linear permutation functions described in the last part of Stage 1 if an omega network is used. If a crossbar (or any other full permutation network) is used, we only need to add two passes.

The operation times for Stage 1 are then:

Fetch : $7\log_2 m-4$

Store : $2\log_2 m-1$

Align : $(\log_2 m)$ $+4\log_2 m+1$

Processor: $\log_2 m(\log_2 m+3)/2$

For Stage 2, the operation times are:

Fetch : $3\log_2(n/m)$

Store : $\log_2(n/m)$

Align : $\log_2(n/m)+\log_2(n/m)\log_2 m$

Processor: $2\log_2(n/m)+\log_2(n/m)\log_2 m$

The total times for this algorithm are then:

Fetch : $3\log_2 n+4\log_2 m-4$

Store : $\log_2 n+\log_2 m-1$

Align : $\log_2 n.\log_2 m+\log_2 n+3\log_2 m+1$

Processor: $\log_2 n.\log_2 m-(\log_2 m)^2/2+2\log_2 n-\log_2 m/2$

3.2.4 <u>Using a Moderate Number of Processors</u>

This algorithm is derived from [24]. For the matrix multiplication, however, this implementation does not use the logsum method. Instead, it uses the more efficient parallel-product serial-sum method. The preliminary discussion of this algorithm will be similar to that of Section 3.2.3.

For each stage $(j+1)$, $(0 \le j < \log(n/m))$, we have $n/2r-1$ $G^{(j)}$'s. We allocate rm processors for each of these G's. For $j=0$ (or $r=m$), we need a total of $(n/2m-1)m^2$ processors. Since we assume that n,m,p are all powers of two, therefore we have $p=(n/2m) x m^2 = mn/2$.

In subsequent description, we will assume $p=mn/2$. If in actual case, $p<mn/2$, then we will apply folding to the algorithm.

$$\text{If } L = \begin{pmatrix} 1 & & & & & \\ (1,3) & 1 & & & & \\ (2,2) & (2,3) & 1 & & & \\ (3,1) & (3,2) & (3,3) & 1 & & \\ (4,0) & (4,1) & (4,2) & (4,3) & & \\ & (5,0) & (5,1) & (5,2) & & \\ & & (6,0) & (6,1) & \ddots & \\ & & & \ddots & \ddots & \ddots \end{pmatrix}$$

In general, let (i,j) be the (m-j)th element on the ith row of L, where $0 \leq i < n$ and $0 \leq j < m$. Then (i,j) will be stored in memory ((i mod m)*m+j) of the $\lfloor i/2 \rfloor$th m^2-partition. For the L matrix given above, the storage map of the first m^2-partition is shown in Figure 3.6.

	Memory						
	0	1	2	3	4	....	14	15
L :	(0,0)	(0,1)	(0,2)	(0,3)	(1,0)	....	(3,2)	(3,3)
	(4,0)	(4,1)	(4,2)	(4,3)	(5,0)	....	(7,2)	(7,3)

Figure 3.6 Storage Map of the First m^2-Partition

The ith element of f is stored in the same memory module as (i,0).

For initialization, we will partition the system into n/2m m^2-partitions. We will solve G_i for i even first, then f_i for i even, then G_i for i odd, and finally f_i for i odd. m multiplies and m additions will be required for each of the calculations. Hence a total of 2m*4 =8m steps will be required for the initialization part.

Before we present the algorithm, we first define a swap in k-partition as a k/2-end-around-shift in k-partition.

Algorithm:

```
Stage 1 (Initialization):

A) All arrays are declared as <n/2m,m,m>.

B) Repeat for h=0,1;
   1. Transfer L(i,j,x),RA=h, to L1(i,j,x), ∀ i,j,
      and ∀ x such that m-j≤x<m.
   2. Left shift by j in m-partition L(i,j,x),RA=h, to
      R(i,j,x),∀ i,j, and ∀ x such that 0≤x<m-j.
   3. Repeat for q=0 to m-1;
      a. Fetch R(i,q,x), ∀ i,x.
      b. Broadcast D(i,q,x) to R1(i,*,x).
      c. Fetch L1(i,j,m-j), ∀ i,j.
      d. Broadcast D(i,j,m-j) to R2(i,j,*).
      e. Multiply R1 and R2 into R3.
      f. Fetch R(i,j,k), ∀ i,j,k.
      g. Transfer D(i,j,k) to R2.
      h. Add R2 and R3 into R3.
      i. Transfer R3 to D.
      j. Store D(i,j,k) into R(i,j,k), ∀ i,j,k.
   4. Repeat for q=0 to m-1;
      a. Fetch f(i,q,0), ∀ i, RA=h.
      b. Broadcast D(i,q,0) to R1(i,*,0).
      c. Fetch L1(i,j,m-j), ∀ i,j.
      d. Left shift D(i,j,m-j) by (m-j) to R2(i,j,0).
```

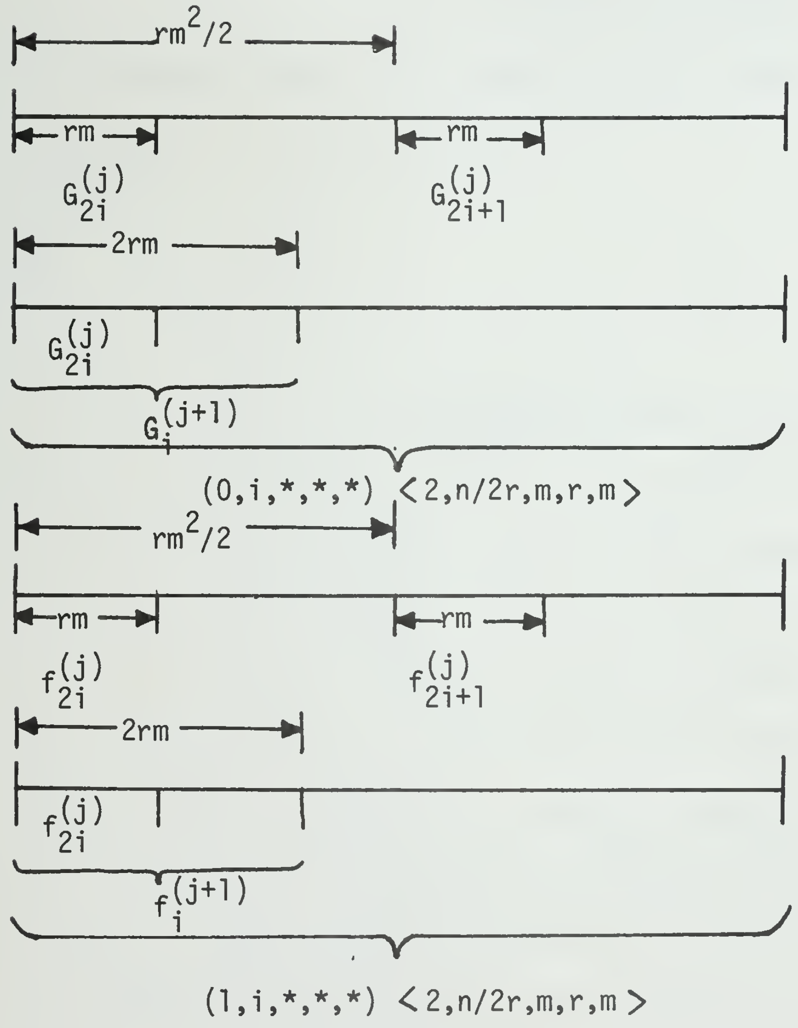


Figure 3.5 Storage Map at Step j of Stage 2

```
        e. Multiply R1 and R2 into R3.
        f. Fetch f(i,j,0), ∀ i,j.
        g. Transfer D(i,j,0) to R2.
        h. Add R2 and R3 into R3.
        i. Transfer R3 to D.
        j. Store D(i,j,0) into f(i,j,0), ∀ i,j.
C) Done.
```

Stage 2:

```
A) Repeat for j=0 to log(n/m)-1;
      1. Set $r=2^j m$.
      2. Declare all arrays as <n/2r,r,m>.
      3. Fetch f(i,k,0),RA=1, ∀ i,k.
      4. Transfer D(i,k,0) to ACC.
      5. Repeat for q=0 to m-1;
         a) fetch R(i,k,q), RA, ∀ i,k.
         b) left shift D by q to R1.
         c) fetch f(i,r-m+q,0), RA=0, ∀ i.
         d) broadcast D(i,r-m+q,0) to R2(i,*,0).
         e) multiply R1 and R2 into R3.
         f) subtract R3 from ACC.
      6. Fetch f(i,j,0),RA=0, ∀ i,j.
      7. Transfer D(i,j,0) to R1.
      8. Transfer R1(i,j,0) to D, ∀ j and ∀ i even.
      9. Swap ACC(i,j,0) in 2rm-partitions to D, ∀ j and
```

```
    ∀ i even.
10. Store D to R, RA=0.
11. Swap R1(i,j,0) in 2rm-partitions to D, ∀ j and
    ∀ i odd.
12. Transfer ACC(i,j,0) to D, ∀ i odd, and ∀ j.
13. Store D to R, RA=1.
14. If $j=\log_2(n/m)-1$, then goto B.
15. Set ACC=0.
16. Repeat for q= 0 to m-1;
    a) fetch R(i,k,q), RA=1, ∀ i,k.
    b) broadcast D(i,k,q) to R1(i,k,*).
    c) fetch R(i,r-m+q,k), RA=0, ∀ i,k.
    d) broadcast D(i,r-m+q,k) to R2(i,*,k).
    e) multiply R1 and R2 into R3.
    f) subtract R3 from ACC.
17. Fetch R(i,j,k),RA=0, ∀ i,j,k.
18. Transfer D(i,j,k) to R1.
19. Transfer R1(i,j,k) to D, ∀ j,k  and ∀ i even.
20. Swap ACC(i,j,k) in 2rm-partitions to D, ∀ j,k and
    ∀ i even.
21. Store D to R, RA=0.
22. Swap R1(i,j,k) in 2rm-partitions to D, ∀ j,k and
    ∀ i odd.
23. Transfer ACC(i,j,k) to D, ∀ i odd, and ∀ j,k.
24. Store D to R, RA=1.

B) Done.
```

<u>Analysis</u>:

All of the alignment functions used in this algorithm can be easily shown to be omega passable, by the simple application of the omega Partition Theorems. Steps 9, 11, 20, and 22 of Stage 2 show the swap operation described earlier in this section.

The total times for this algorithm are:

Fetch: $\log_2(n/m)(4m+3)$

Store: $4\log_2(n/m)+4m-2$

Align: $\log_2(n/m)(4m+4)+6m$

Processor: $4m.\log_2(n/m)+6m$

3.3 Matrix Multiplication on a Parallel Processing System

A Fortran code section that performs matrix multiplication is as follows:

```
        DO 10 I=1,N
        DO 10 J=1,N
        DO 10 K=1,N
   10   A(I,J)=A(I,J)+B(I,K)*C(K,J)
```

An efficient way to perform the calculation would be to compile the product by rows (parallel on J) as shown below.

```
        DO 10 I=1,N
        DO 10 K=1,N
   10   A(I,*)=A(I,*)+B(I,K)*C(K,*)
```

This algorithm will require $O(N^2)$ shifts to align the operand matrices. A one-stage perfect shuffle network simulating an omega network will take $\log_2 N$ steps per shift, and the Illiac IV type of switch will take $O(\sqrt{N})$ steps per shift on the average. So a total of $O(N^2 \log_2 N)$ or $O(N^2 \sqrt{N})$ routing steps are required for matrix multiplication. However, using the algorithm which follows, we need only $O(N^2)$ steps.

We first need to define the following two notions:

<u>Notation</u>: If G is a permutation of some input set, G^i

implies i consecutive applications of the permutation G to the input set.

<u>Definition</u>: A G-permutation is defined as a permutation G such that $G, G^2, G^3, \ldots, G^N$ are distinct and form a group with $G^N = I$, the identity permutation.

Every G permutation can be uniquely represented as a cycle $(i_0, i_1, \ldots i_{N-1})$ where $G(i_0)=i_1, G(i_1)=i_2, \ldots, G(i_{N-1})=i_0$.

Two obvious G-permutations are the +1 shift permutation and the -1 shift permutation. In general, +k shift and -k shift permutations will be G-permutations if k is relatively prime to N. Some nonshifting G-permutations can be found using a perfect shuffle based permutation. The G-permutations have a general form of:

$$G(i) = [2i+b(i)] \bmod N,$$
$$\text{where } b(i) = b(i+N/2) \ \forall\ i=0\ldots N/2-1,$$
$$\text{and } \ b(i) = 0 \text{ or } 1 \ \forall\ i.$$

A list of all $\{b(i), i=0\ldots.N/2-1\}$ that will give G-permutations for N=4 and 8 and the corresponding G-permutations are listed in Table 3.1.

Size	b(i)	G-permutation
4	1 1	(0 1 3 2)
8	1 1 0 1	(0 1 3 7 6 5 2 4)
	1 0 1 1	(0 1 2 5 3 7 6 4)

Table 3.1

Assume we want to multiply two matrices A and B to form C and that they are all of size NxN. The first method uses N processors and requires that the storage scheme for the matrices be 1-skew and 1-skip. The storage pattern is shown in Figure 3.7. Each processor will have a corresponding memory from which it can fetch data. Any data a processor wants but not in its own memory will have to be routed from the other processors. This algorithm also calculates the relative address (RA) for each array it references.

Memory			
0	1	2	3
(0,0)	(0,1)	(0,2)	(0,3)
(1,3)	(1,0)	(1,1)	(1,2)
(2,2)	(2,3)	(2,0)	(2,1)
(3,1)	(3,2)	(3,3)	(3,0)

Figure 3.7 1-skew 1-skip Storage Scheme

Each processor has a wired-in processor port number, PPN ($0 \leq$ PPN $<$ N-1). T is a temporary array.

Algorithm:

```
A) Repeat for IC = 0 to N-1;
   1. fetch A, RA=IC, into R1.
   2. set IR = (PPN-IC) mod N.
   3. repeat for IT = 0 to N-1;
      a. fetch B, RA = IR, into R2.
      b. multiply R1 and R2 into R3.
      c. G-permute IR.
      d. store T, RA=(PPN-IR,mod N) from R3.
      e. G-permute R1.
   4. set R1=0.
   5. repeat for IT = 0 to N-1;
      a. fetch T, RA=IR, into R2.
      b. add R1 and R2 into R2.
      c. G-permute R1.
      d. G-permute IR.
   6. store C(RA=IC) from R1.
B) Done.
```

The significance of this result is that for certain one stage networks, if there exists a G-permutation, then each intermediate routing will take only O(1) time instead of

$O(\log_2 N)$ time or $O(\sqrt{N})$ time. This greatly reduces the alignment time for the system.

There are many variations of this algorithm, each for a different memory skewing scheme. Two of them can be used for a parallel processing system with twice the number of memory modules. They will be presented in Appendix A.

There is another algorithm which uses N^2 processors. However, it works only for a $N^2 x N^2$ network with $\pm N$ shift and ± 1 shift connections. This algorithm takes a total of N+2 memory fetches and N+1 memory stores . The total number of alignment requests is 3N and the total number of arithmetic operations is 2N. Hence the alignment time matches in order of magnitude with the memory and arithmetic operations.

The initial storage scheme is simple. All matrices are stored in a linear manner, i.e., element (i,j) will be stored in memory (Ni+j).

Algorithm:

```
A) Fetch B into R1.
B) Repeat for j =0 to N-1;
   1. store R1 into T, RA=j.
   2. left shift R1 by N.
C) Fetch A into R1.
D) Set ACC=0.
```

```
E) Repeat for j =0 to N-1;
   1. Fetch T, RA=(PPN+j)modN, into R2.
   2. multiply R1 and R2 into R3.
   3. add R3 to ACC.
   4. right shift R1 by N into R2.
   5. transfer R2(i,0)<N,N> to R1(i,0)<N,N>.
   6. left shift R1 by 1.
F) Store ACC into C.
```

The above two algorithms show how computations can be tailored to fit a simple network so as to minimize the routing times.

4. PROCESSOR SYSTEM SIMULATION TECHNIQUES

4.1 Introduction

In order to evaluate the true effectiveness of a parallel architecture, we must hypothesize a compiler capable of compiling ordinary programs into code which most effectively utilizes the architecture, especially the data alignment capabilities. The resulting code could then be simulated and the important performance measures determined. This is the objective of our Analyzer/Simulator project. It involves the simulation of program execution on some proposed parallel processing systems. The front end of this project is a program analyzer which accepts Fortran source programs, and by detailed analysis of the control and data dependencies it produces a highly parallelized version of the original program (see [26]). Next, this parallelized version is input to another program, the Resource Request Generator(RRG), which attempts to compile the parallelized program into simulatable code. The code is a set of machine resource requests with data dependencies embedded in it. A machine resource can be a scalar or array processor, an alignment network, or the whole bank of array memories. The task of the RRG is to decide on the best way to slice the computation specified by each instruction node, based on the size of the matrices, the number of available processors, the matrix storage scheme, and the type of alignment network. Finally, the output of the RRG is input to a simulator capable of simulating a wide variety of architectures. Here the time required, utilization of various resources, and speedup and efficiency of the program's execution in the given parallel processing system will be calculated. Machine organization parameters can be specified by the user. These parameters include the storage scheme, the alignment network, the processor and memory speeds, the number of array memories, and the number of processors in the array processor system.

A block diagram showing the general organization of the software is shown in Figure 4.1. The Program Analyzer is described elsewhere [26] and we will not discuss it here. In this section we will describe the RRG and machine simulation. Some experimental results will also be presented. In Section 4.2 we will discuss the input data structure and available machine parameters for the RRG. In Section 4.3 we will describe the output of the simulator in the form of performance measures. Then in Sections 4.4 through 4.6, some of the algorithms and strategies of the RRG will be described. Finally, in Section 4.7, we will discuss some of the preliminary results of the initial set of experiments.

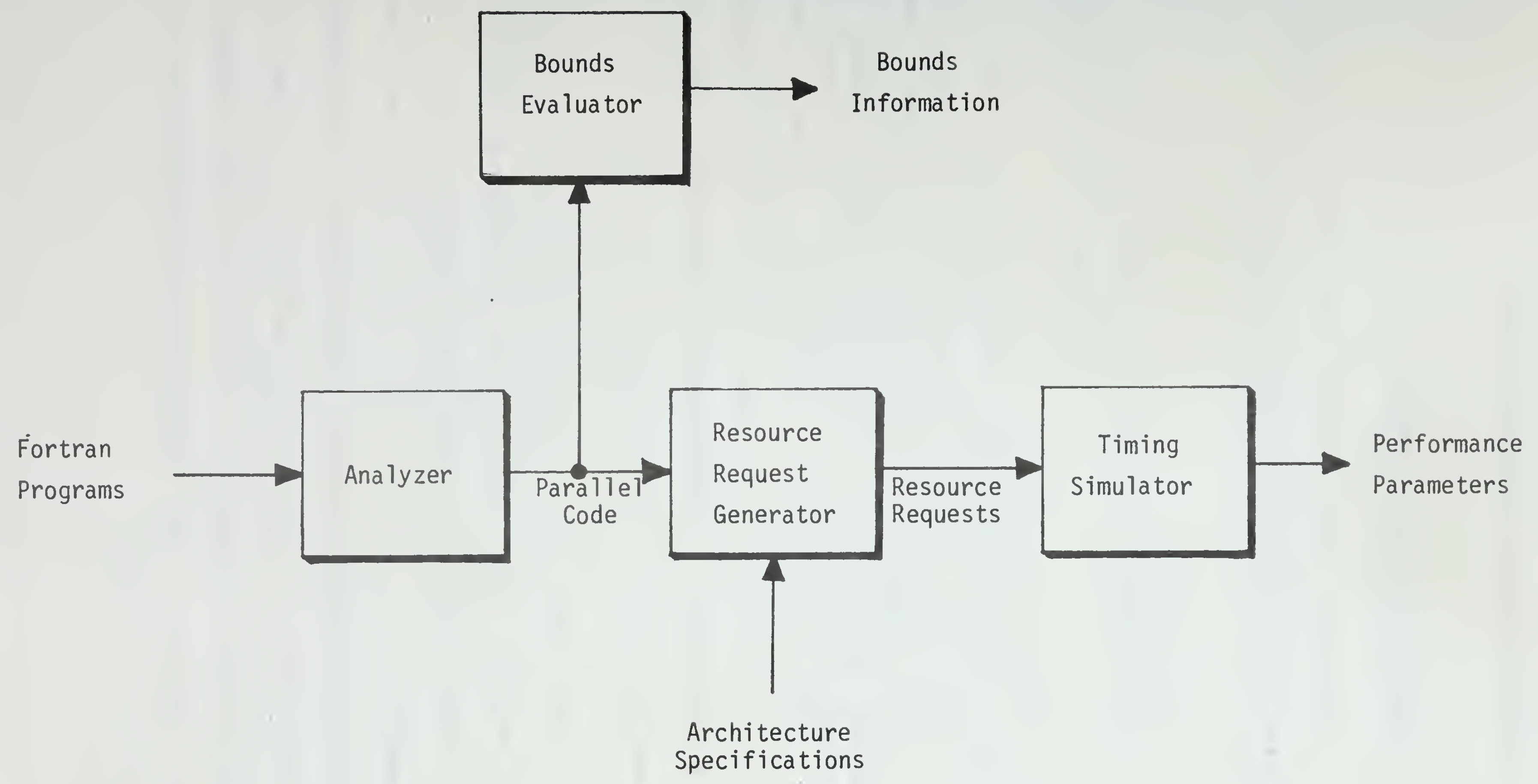


Figure 4.1 Analyzer/Simulator Organization

4.2 Simulator Input Specifications

4.2.1 Input Instruction Nodes

The most easily recognizable form of parallelism is typified by a matrix addition shown below.

```
        DO 10 I=1,N
        DO 10 J=1,M
   10   A(I,J)=B(I,J)+C(I,J)
```

The Program Analyzer will determine what the dependency limitations are for each program segment (in this case, there are none), and then break them into machine-code-like instruction nodes. Each instruction node will provide all the information concerning the operator, the two operands and the result.

After the Fortran Analyzer phase, all parallel DO loop indices are distributed into each instruction node. The DO loop limits are normalized to start with 0 and have increments of 1 only. We first assume that there are n active DO Loop indices in a particular instruction. The j-th DO loop index, I_j, may have an upper limit, U_j, as a function of $I_1, I_2, \ldots, I_{j-1}$. Assuming the function is a linear function, the U_j's can be represented by a nx(n+1) matrix, D, such that

$$\begin{bmatrix}U_1\\U_2\\ \cdot\\ \cdot\\U_n\end{bmatrix} = n\updownarrow \underset{\longleftarrow\ n+1\ \longrightarrow}{\begin{bmatrix} & & c_1\\ & & c_2\\ & D & \cdot\\ & & \cdot\\ & & c_n\end{bmatrix}} \begin{bmatrix}I_1\\I_2\\ \cdot\\ \cdot\\I_n\\1\end{bmatrix}$$

Note that except for the last column, the D matrix is strictly lower triangular.

In a similar fashion, each k-dimensional array (with linear subscripts) being referenced in a node with n active DO loop indices, will have a corresponding k by (n+1) coefficient matrix C. Let E_j be the subscript expression of the jth dimension, then

$$\begin{bmatrix}E_1\\E_2\\ \cdot\\ \cdot\\E_n\end{bmatrix} = k\updownarrow \underset{\longleftarrow\ n+1\ \longrightarrow}{\begin{bmatrix} & & \\ & & \\ & C & \\ & & \\ & & \end{bmatrix}} \begin{bmatrix}I_1\\I_2\\ \cdot\\ \cdot\\I_n\\1\end{bmatrix}$$

<u>Definition</u>-- Let there be M memory units. A p-ordered N-vector (mod M) is defined as a vector of N elements whose i-th logical element is stored in memory unit pi+c (mod M) where c is an arbitrary constant.

The idea of a p-ordered N-vector is very useful in finding the number of cycles required to access a vector or to align it using certain alignment networks.

Using a generalized skewing scheme as in [Lawrie 1], for an array with k dimensions, we will have $(m_1, m_2, \ldots, m_k)$ skewing. Assuming an array operand in an instruction node has k dimensions and n active indices, then we define an order vector, V, of n+1 elements as:

$$\overset{\longleftarrow n+1 \longrightarrow}{\begin{bmatrix} \quad V \quad \end{bmatrix}} = \overset{\longleftarrow \quad k \longrightarrow}{\begin{bmatrix} m_1 & m_2 \ldots m_k \end{bmatrix}} \overset{\longleftarrow n+1 \longrightarrow}{\begin{bmatrix} \\ \\ \qquad C \qquad \\ \\ \\ \end{bmatrix}} \updownarrow k$$

For an array element defined by any particular set of values $\{I_1 = h_1, I_2 = h_2, \ldots I_n = h_n\}$, the element will be stored in memory port z, where

$$z = [\quad V \quad] \begin{bmatrix} h_1 \\ h_2 \\ \cdot \\ \cdot \\ h_n \\ 1 \end{bmatrix}$$

In addition, the importance of the order vector lies on the fact that for any partition of the array formed by running the jth active index parallel, the partition is a V_j-ordered vector (mod M), where M is the number of memories. When the order and number of elements of a partition are calculated, the number of cycles required to access and align the vector can be easily determined.

For Fortran statements that cannot be easily dispatched as array or scalar operations, they will be grouped as recurrences nodes. Each node represents a R<n,m> system (c.f.[22]). Each R<n,m> system will be broken into as many smaller recurrence units as possible. Information such as the number of smaller recurrence units and the values of n and m for the units can be found in a recurrence node. With this information, we can determine which is the best recurrence solving algorithm to use and its corresponding execution time.

4.2.2 Machine Parameters

In the parallel architecture that we simulate we assume that the resources can operate in an overlapping (or pipelining) fashion. However, we still honor the dependency between different instruction nodes. Each resource will have its own resource queue to hold the waiting requests. Hence one node may be using the alignment network while an independent node can start fetching its operands from the memory system.

It is impossible to simulate every known parallel architecture. So we concentrate on two classes of architectures. The first class is shown in Figure 4.2 and the second in Figure 4.3. Note that the one in Figure 4.2 resembles that assumed in Chapter 3. The second type has two alignment networks, one for input to the processing system and the other for output to the memory system. This class can be chosen by setting the parameter option M_PARAM.TWO_AL_NET to 1.

The scalar memory and scalar processor in Figure 4.2 and 4.3 are optional and can be chosen by setting M_PARAM.SM and/or M_PARAM.SP to 1's.

The number of processing elements in the processing array and the number of memories can be selected using the parameters M_PARAM.NUM_PROC and M_PARAM.NUM_MEM.

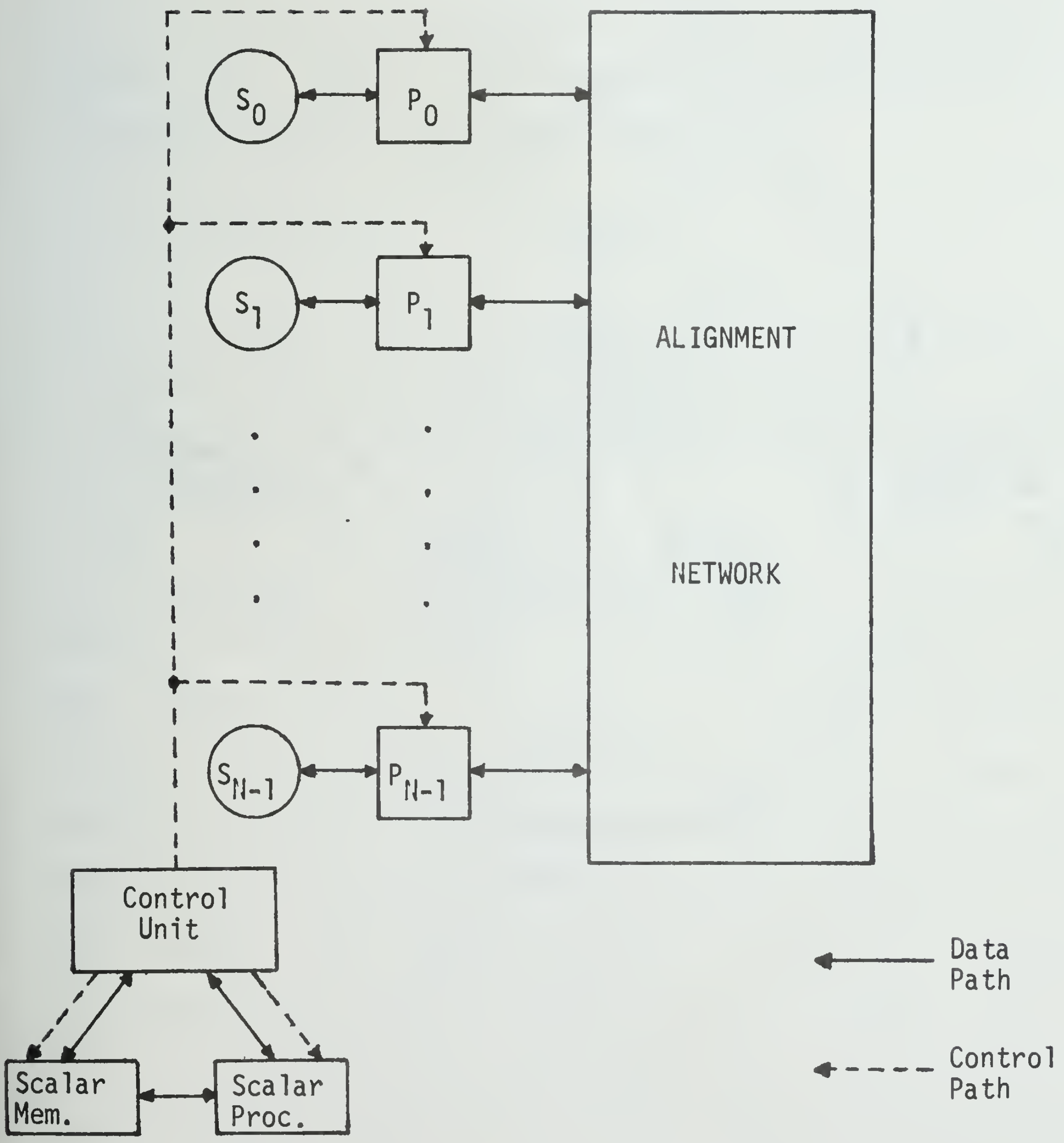


Figure 4.2 Machine Configuration A

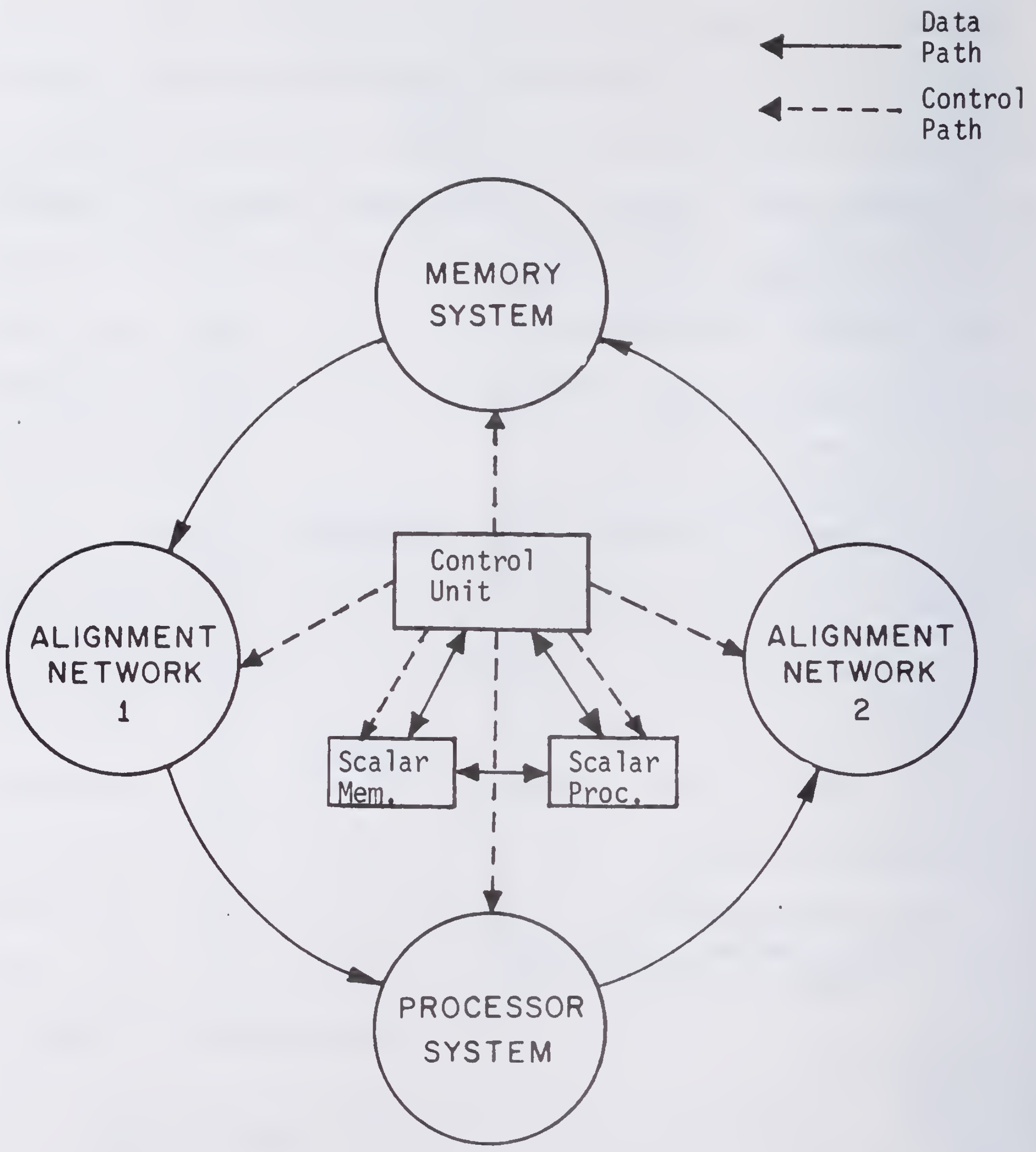


Figure 4.3 Machine Configuration B

The skewing system chosen can be specified by assigning values to the array M_PARAM.MEM_MAP. For example to get (1,1) skewing, we would put the numbers 0,0,0,1,1 into the MEM_MAP array.

As for the alignment network, right now we can choose any one of the four possible networks by setting M_PARAM.AN_TYPE to the appropriate value.

AN_TYPE	= 1	:	crossbar
	= 2	:	omega network
	= 3	:	$\pm\sqrt{p},\pm 1$ shift network
	= 4	:	± 1 shift network

The memory cycle time can be specified by using M_PARAM.MCYCLE_TIME, and the scalar memory time be specified using M_PARAM.S_MEM_TIME. To allow for pipelining, we have two separate time fields for each resource request. The first is RT which contains the time that must lapse before another request for the same resource can be started. The other is IT, which contains the time required to finish processing the request. If a particular resource is pipelined, then RT will be the pipeline segment time and IT will be the length of the whole pipe. In this case, IT is greater than or equal to RT. For a memory request, however, RT will be the cycle time and IT will be the access time. In this case, IT is less than or equal to RT.

The alignment times are specified by M_PARAM.A_CT_IN and M_PARAM.A_CT_OUT. The processing times can be assigned by the user using the array M_PARAM.OP_TIME. The elements are times required for simple assignment, addition, subtraction, multiplication, and division, respectively. We also allow the users to define their own built-in function and user defined function times in M_PARAM.BUILTIN_TIME and M_PARAM.USERFCN_TIME respectively.

A sweeping index is defined to be the active index that is to be run parallel in order to produce the desired partition. One option that the user has is tc declare what he wants as the sweeping index. The other is to let the Simulator choose the best index in terms of execution times. To choose the first option, we will have to set M_PARAM.SWPOPT to 0 and to set the array M_PARAM.SWEEP_INDX to the running indices required.

For standard algorithmic procedures, such as recurrence handling, the operation times for various resources have been calculated in Chapter 3. Thus we just need to substitute in the formulas for the cperation times rather than perform detailed simulation of the algorithm. However, we will be missing certain overlap parallelism. This overlap can be assigned by the user using M_PARAM.CVERLAP. In appropriate cases, IT will be set to OVERLAP*RT/100, and will be less than RT.

In this section, we have discussed various options that are available to the users. By setting all the appropriate options, the user has defined a machine configuration that he wants to study. In the next section, we will discuss the outputs available from this Simulator.

4.3 Simulator Outputs

The output of the simulator is a set of performance measures. One such measure is T_p, the time required for simulated execution of the program graph from the Program Analyzer in the specified machine organization using p processors. If T_1 is the execution time for the same program graph, then we define another measure, the speed factor, F_p, as T_1/T_p. In addition, the Simulator calculates measures of the utilizations of various system resources. The utilization of each resource is broken down into several separate utilizations, U_a, U_s and U_{IF}. First, U_a, the array duty cycle, is the percentage of time that at least one processor is performing a computation. However, whenever an array operation is being performed, only some of the processors may be actually doing useful work. This is measured by the slicing utilization, U_s. For example, to add two 30 element vectors together using 20 processors would require two steps. The first step would form the first 20 sums and would use all 20 processors resulting in a slicing utilization, U_s, of 100%. The second step would form the last 10 sums using only 10 processors and would result in U_s=50%. The overall U_s would then be 75%. Finally, some processors are turned off because of IF statements in the original programs, and this is measured by U_{IF}. For example, assume that in the following program, 1/3 of the B(I) are less than zero:

```
      DO 10 I=1,30
10    IF (B(I).GE.0) A(I)=A(I)+B(I)
```

Then U_{IF} =67%. Thus, using 20 processors on this program, U_a might be 80%, for example, because the processors are waiting for memory access or data alignment. Of this 80% of the time, only 75% of the processors could be used because of the difference between the number of processors and the array size (U_S=75%), and of these 75% of the processors, only 67% are turned on (U_{IF} =67%). Thus, the total average processor duty cycle, U_T, is equal to U_a*U_S*U_{IF} = 80%*75%*67% = 40%. By separating the components of processor utilization in this way we can determine the source of processor inefficiencies.

4.4 Sweeping Indices

As described in Section 4.2.2, when an instruction node can be swept by more than one index, there are two options for the user to define the sweeping index. One is to specify it in the parameter M_PARAM.SWEEP_INDX. The other is to let the RRG choose the best index for each individual node. When there are other indices having upper limits that depend on the sweeping index, we need to modify the C and D matrices before the sweeping is allowed. In other words, an instruction node can be swept on an index I_i if and only if, in the D matrix, the i-th column has all zeroes.

For example, if the instruction node looks like:

```
DO   I = 0,N-1
DO   J = 0,I+k
  A(I,J) = 0
```

then $C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ and $D = \begin{pmatrix} 0 & 0 & N-1 \\ 1 & 0 & k \end{pmatrix}$

To sweep on index I, we need to expand the node to two nodes:

```
DO  J = 0,k-1                      DO  J = 0,N-1
DO  I = 0,N-1          and         DO  I = 0,N-J-1
  A(I,J)=0                           A(I+J,J+k)=0
```

Now there are two sets of C and D matrices. They are:

$$C_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$D_1 = \begin{bmatrix} 0 & 0 & N-1 \\ 0 & 0 & k-1 \end{bmatrix}$$

$$C_2 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & k \end{bmatrix}$$

$$D_2 = \begin{bmatrix} 0 & -1 & N-1 \\ 0 & 0 & N-1 \end{bmatrix}$$

Note that after this transformation, the first column of D_1 and D_2 are all zeroes. Hence the two transformed nodes can be swept on index I.

In general, given a node

```
DO I=0,N-1
DO J=0,hI+k
  A(I,J)=0
```

and we want to sweep on index I, the original loop will have to be transformed into:

```
DO J=0,hN-h+k
DO I=0,f(J)
  A(I,J)=0
```

The first problem here is what should be the equation for f(J) in general. If h is not equal to 1, f(J) can contain many modulo functions, which are nonlinear, and cannot be represented easily in our linear D matrices. Another problem is that if h is large, many of the vectors A(*,J) will be small vectors which can seriously degrade the efficiencies of a parallel system. So the solution we picked is to do this kind of transformation only if h=1.

Consider a more general case than the one shown above:

```
DO I=0,N-1
DO J=0,I+k
A(pI+qJ+r,xI+yJ+z)=0
```

i.e. $C = \begin{bmatrix} p & q & r \\ x & y & z \end{bmatrix}$

and $D = \begin{bmatrix} 0 & 0 & N-1 \\ 1 & 0 & k \end{bmatrix}$

We can split the node into two parts:

```
DO I=0,N-1
DO J=0,k-1
A(pI+qJ+r,xI+yJ+z)=0
```

and
```
DO I=0,N-1
DO J=0,I
A(pI+qJ+r+qk,xI+yJ+z+yk)=0
```

The first part thus has:

$$C_1 = \begin{pmatrix} p & q & r \\ x & y & z \end{pmatrix} = C$$

and $$D_1 = \begin{pmatrix} 0 & 0 & N-1 \\ 0 & 0 & k-1 \end{pmatrix}$$

The second part is equivalent to:
```
DO J=0,N-1
DO I=J,N-1
A(pI+qJ+r+qk,xI+yJ+z+yk)=0
```

and is also equivalent to:

```
DO J=0,N-1
DO I=0,N-J-1
A(pI+(p+q)J+r+qk,xI+(x+y)J+z+yk)=0
```

or $$C_2 = \begin{pmatrix} p & p+q & r+qk \\ x & x+y & z+yk \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & -1 & N-1 \\ 0 & 0 & N-1 \end{pmatrix}$$

Hence $C_2 = C * X$

where $X = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & k \\ 0 & 0 & 1 \end{bmatrix}$

Therefore, $V_1 = M * C_1$ and $V_2 = M * C_2$ will form a pair of order vectors that can determine if the matrix can be swept by the index I.

In general, if we reduce U_j from I_h-dependent to I_h-independent, X will be a $(n+1) \times (n+1)$ matrix with ones on the diagonal, another 1 at position (j,h) and a k at position $(h,n+1)$.

4.5 Array Slicing

When an instruction node represents a larger operation than the processor system can handle, the array operands in the node have to be sliced. Let us define the required number of slices as S. The slicing utilization, U_S, discussed in Section 4.3, is defined as the percentage of the amount of a resource that is being utilized. These are the two most important quantities to be discussed in this section.

When the upper index bounds are all independent (i.e. the first n columns of D are all 0's), it is easy to find S and U_S:

$$S = \lceil N/p \rceil \prod_{\substack{i=1 \\ i \neq I_S}}^{n} [D(I,n+1)+1]$$

$$U_S = N/(\lceil N/p \rceil * p) * 100\%$$

where I_S is the sweeping index, $N=D(I_S,n+1)+1$, and p is the number of processors.

After transforming the loop as discussed in Section 4.4, no upper index bound will be dependent on I_S. If, however, the upper bound of I_S depends on index h, we have to calculate S and U_S differently. If we have this kind of instruction node:

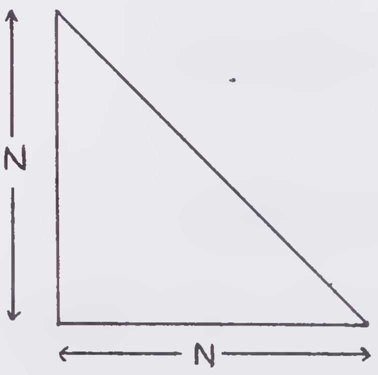
```
DO h=0, N-1
DO I =0, ah+b
instruction
```

a rough estimate of S can be calculated as follows: the average upper bound of I_S is aN/2+b. Hence

$$S = \lceil (aN/2+b)/p \rceil * N$$

and $$U_S = (aN/2+b)/(S*p)*100\%$$

If a=1, we have an upper triangular system and we can find more accurate values for S and U_S. Let us first consider S for a purely triangular system, as shown below:

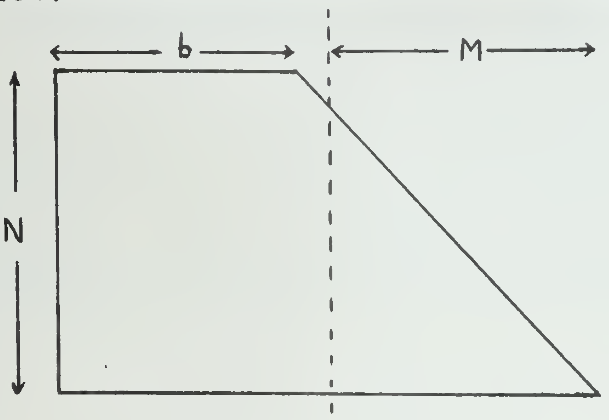


Breaking it into $\lceil N/p \rceil$ sets of columns. The first set will contribute N to S. The second set will contribute (N-p) to S, and so on. So

$$S = \sum_{i=0}^{\lceil \frac{N}{p} \rceil - 1} (N-pi)$$

$= \lceil N/p \rceil \{N-p(\lceil N/p \rceil -1)/2\}$.

Now let us return to the original triangular system, as shown below:



where $M=N+b-\lceil b/p \rceil * p = N-(p-b) \bmod p$.

The first half contributes $N* \lceil b/p \rceil$ to S. The second half is a purely triangular system with size MxM, and thus contributes $\lceil M/p \rceil \{M-p(\lceil M/p \rceil -1)/2\}$ to S. Therefore

$S=N \lceil b/p \rceil + M/p \ \{M-p(M/P -1)/2\}$.

The total number of elements = $N(N+1+2b)/2$. Hence

$U_S = N(N+1+2b)/(2Sp)*100\%$.

When S is greater than one, we will have to replicate the resource requests S times. However, in order to save simulation time, we will devise the following procedure.

We first observe that in general, each slice will follow the same general pattern: a fetch, an alignment, a processor cycle, then another alignment and finally a store. When there are more slices, they will be duplicates of the sequences, but with slight time displacements, like:

```
F

A   F

P   A   .

A   P   .   .

S   A       .   F

    S           A

                P

                A

                S
```

The middle part of the operation will be the concurrent operations of F-A-P-A-S, each working on a different slice. It is repeated (S-4) times. To expedite simulation, we put an implied DO loop around this middle part. This DO loop will be simulated repeatedly until no other request node is using any system resource. Then one more iteration will be done to figure out the iteration time. This time is then multiplied by the number of remaining iterations to find the total time. This method will reduce the amount of parallelism slightly; however, it reduces the simulation time greatly.

Only one DO loop can be active at any time for any one level specified, in order for the above simulation method to work. This can be achieved by generating a resource request (for that level) at the DO node. The resource will be released at the END node, when the required iterations have been finished. This effectively locks out any other independent DO loop activity which would interfere with timing the "last" iteration. After the resource is released, another DO can be activated by being granted that resource.

4.6 Resource Time Calculation

After we have found S and various utilizations, we will proceed to find the resource times of various needed resources for that particular instruction node. Scalar memory and processor times are simple to calculate and we will not elaborate on these. However, recurrence and ordinary vector operations need further explanation. Shift networks present a different set of calculation and will be treated in a separate section.

4.6.1 Recurrence Handling

For recurrence nodes, we have analyzed in Chapter 3 the conditions under which certain recurrence solving algorithms should be used. We have also found the corresponding resource times for each algorithm. Hence, we can save a lot of simulation time by simply putting in the corresponding resource times when a recurrence node is encountered. This way, we assume once a recurrence node is encountered, we will preempt the machine to do just the recurrence. Usually there is overlap between various resource times, i.e., the sum of all the resource times should be greater than the total execution time. To account for this effect, we set the total execution time to a constant parameter, OVERLAP, multiplied by the sum of the resource times. This OVERLAP can be found by first writing the recurrence solving algorithm in Fortran and then running it through the Analyzer/Simulator. The average OVERLAP is calculated for various array sizes and machine configurations. This method will not give us the true value for the execution time of each recurrence calculation. However, it will give us a moderately reliable estimate.

4.6.2 Vector Operations

(using crossbar or omega network)

For a vector operation node using crossbar or omega networks, the resource times are easy to calculate after the order vectors (described in Section 4.2.1) for the operands are calculated. For memory accesses, if the order for that particular operand on a particular sweep is p, then consecutive elements can be found p memory modules apart. Hence we need g=gcd(p,M) memory fetches before we can fetch the entire slice. In general, the number of memory cycles required to access a p-ordered N-vector (defined in Section 4.2.1) stored in M memories = $\lceil N*g/M \rceil$. After each memory cycle, the time required for aligning a p-ordered vector using a crossbar and for aligning before storing into a p-ordered vector using omega network is equal to 1 network cycle. Nevertheless, to fetch a p-ordered vector slice using the omega network, we also need g network cycles. Hence the corresponding total number of network cycles are:

1) crossbar -- $\lceil N*g/M \rceil$.

2) omega -- $g\lceil N*g/M \rceil$ for fetching,

$\lceil N*g/M \rceil$ for storing.

4.6.3 Illiac Type Shift Networks

For processing systems with N processors, if the interprocessor connections are $\pm\sqrt{N}$ and ± 1-shifts, we call the alignment network the Illiac type shift network.

For this type of network, when a uniform shift permutation is requested, the resource time is easy to calculate. Let s be the shift distance required. We first set ss=min(s,N-s), where N is the number of processors. Also let n be $\sqrt{N}$. The shift time required

$$= \lfloor ss/n \rfloor + (ss \bmod n) \qquad \text{for } (ss \bmod n \le n/2),$$
$$= \lfloor ss/n \rfloor + n + 1 - (ss \bmod n) \qquad \text{for } (ss \bmod n > n/2). \qquad (4.1)$$

When a triangular type array is accessed, and the shift distance for each slice is different, we have to find the average shift time for the array. Let A(x) be the average shift time for shift distances 1,2,...,x. If x=Nk+w where N is the number of processors, then

$$A(x) = \frac{NkD(N)+wD(w)}{Nk+w} \qquad (4.2)$$

where D(w) is the average shift time for shift distances 1,2,....,w, (w≤N).

We will first calculate wD(w).

Let d=D(n). We first observe the regular pattern of the shift time for s=1,2,....n.

For n=8, shift dist.	0	1	2	3	4	5	6	7	8
shift time	0	1	2	3	4	4	3	2	1

Summing this arithmetic series, we will get

$$nd = 2*(n/2)(n/2+1)/2 = N/4+n/2, \quad \text{for } n \text{ even,}$$

and
$$= 2*((n+1)/2)((n+1)/2+1)/2-(n+1)/2$$
$$= (n+1)^2/4, \quad \text{for } n \text{ odd.} \qquad (4.3)$$

To observe the shift time pattern, we will show the shift time for various s using N=16 processors.

shift dist.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
time	0	1	2	2	1	2	3	3	2	3	3	2	1	2	2	1

Let us first concentrate on the shift time after we arrive at the required n-partition.

We let $z=\lfloor w/n \rfloor$ and $y=w \bmod n$. z will be the n-partition number while y is the number within a particular n-partition. We set $y=y+1$ for $z \geq n/2$ to account for the extra shift to reach the other end of the n-partition.

$$yD(y) = y(y+1)/2 \quad \text{for } 0 \leq y \leq n/2,$$

$$yD(y) = (n/2)(n/2+1)/2+\sum_{i=\frac{n}{2}+1}^{y}(n+1-i) \quad \text{for } y>n/2,$$

$$= ny-y(y-1)/2-N/4 \qquad (4.4)$$

We then calculate the times required to get to the respective n-partitions. They are:

t_i : $0, 1, \ldots, (n/2-1), (n/2-1), (n/2-2), \ldots, 2, 1.$

Let w*f be the total number of n-shifts we have to do for all s in the first z n-partitions.

$$wf = n\sum_{i=1}^{z} t_i = nz(z-1)/2 \qquad \text{for } 0 \le z \le n/2,$$

$$= n(n/2)(n/2-1)/2 + n \quad (n-i) \qquad \text{for } n/2 < z < n,$$

$$= Nz - nz^2/2 - nz/2 + nN/4. \qquad (4.5)$$

The total number of n-shifts required for s in the n-partition where w is located is equal to q, where

$$q = \min(z, n-1-z) * y \qquad \text{for } z < n,$$

$$= 0 \qquad \text{for } z = n. \qquad (4.6)$$

Hence $wD(w) = znd + yD(y) + wf + q$.

For $w \ge N/2$, we need to subtract $n/2$ from $wD(w)$, since we have overcounted the shift time at $N/2$.

i.e. $$wD(w) = znd + yD(y) + wf + q \qquad \text{for } w < N/2,$$

$$= znd + yD(y) + wf + q - n/2 \qquad \text{for } w \ge N/2. \qquad (4.7)$$

Now we want to find D(N). For $w=N$, $z=n$ and $y=0+1=1$.

Hence $$ND(N) = n(N/4+n/2) + n - 1/2 + 1/2 - N/4 + nN - nN/2 - N/2$$
$$- nN/4 + 0 - n/2$$

$$= nN/2 - N/4 - n/2 \qquad (4.8)$$

Now A(x) in (4.2) can be found easily once ND(N) and wD(w) are known.

If a random shift is required, then an average time of $\sqrt{N}/2$ will be used. If a broadcast function is needed, then we will use the worst case result of $\sqrt{N}$.

When the permutation is other than shift or broadcast, we will apply Orcutt's result of $8(\sqrt{N}-1)$ for omega passable permutations[27].

4.7 Experimental Results

Our initial experiments will deal with the effects of the following architectural parameters:

1) The number of array processors, and the speed of the processors relative to the array memory system. Initially, the processors will be restricted to a single group of processors operating from a single instruction stream (SIMD).

2) The presence or absence of an independent scalar processor and/or memory. The absence of a scalar processor forces scalar operations to be performed by the array processors.

3) The memory system, including the array storage scheme (1-skew, etc.), and the number of memories(power of two or prime).

4) The type of alignment network:
   a) Crossbar,
   b) omega network,
   c) $\pm 1, \pm\sqrt{p}$ shifter (Illiac IV),

These parameters will be studied for a large variety of application programs, and in addition the size of the application programs (i.e. the array sizes) will be varied in order to produce families of performance figures.

The tables below present some preliminary results of experiments on three programs. We would like to stress at this point that these results are preliminary. The three programs can hardly be construed as representative of any large population of applications. The first program, ADVV, is a 4-point relaxation scheme. ADVV was chosen because of its highly parallel nature. The second program, ELMBAK, forms the eigenvectors of a real matrix by back transforming those of the corresponding upper Hessenberg matrix. ELMBAK is reasonably complicated, but has no recurrences. The third program, SLEQ1, is a Gauss-Jordan reduction program. SLEQ1 was chosen because it contains a recurrence relation (a R<18,1> system). We present the results of these three programs only as an indication of the types of results we expect from our experiments, and an illustration of how to interpret the results.

The complete tables of experimental results for these three programs are shown in Appendix E. Some of the more interesting figures are grouped together in Tables 4.1 through 4.4.

Table 4.1 shows the speed factor, $F_P=T_1/T_P$, and processor utilization U_T using 16 processors, 17 memories, a crossbar alignment network, skewed storage, and separate scalar processor and memory. The results are presented as a function of N, the data array sizes. Notice for ADVV, the

Program	N=10	N=16	N=40	N=60	N=100
ADVV	9.7, 43%	16.0, 70%	14.1, 62%	15.9, 70%	16.2, 71%
ELMBAK	2.0, 6%	3.0, 10%	5.9, 23%	8.0, 32%	9.6, 39%
SLEQ1	4.5, 14%	6.8, 21%	9.6, 30%	14.5, 45%	--

Table 4.1 Speed (F_{16}) and processor utilization (U_T) using 16 processors, 17 memories (cf [11]), crossbar alignment network and skewed storage. N is the data array size .

speed factor quickly approaches the maximum value of 16. Processor utilization ranges from 43% to 71%. The result for N=16 indicates that U_a = 70% (U_s=100% since N=p=16 and for ADVV, U_{IF} =100%). Thus,the processors are only busy 70% of the time due to non-perfect overlap of array processor operations with alignment, memory, and scalar operations. However, the speed factor is 16 which would indicate a similar degree of non-perfect overlap in a comparable serial processor. The other programs, ELMBAK and SLEQ1 indicate much lower speed factors and utilizations. SLEQ1 contains recurrences, which are handled in parallel but much less efficiently than the pure vector operations in ADVV. Notice, however, that even though SLEQ1 contains a recurrence, the speed factor of 14.5 is very close to the maximum of 16 when N is 60. We believe it is significant that we are able to handle recurrences this well.

The reason the ELMBAK results are so low illustrates an interesting situation. At the present time programs are compiled into three address vector or scalar instructions. If the vectors are of sufficient length, then an implicit loop is established in order to cycle the processors, memories, etc. a sufficient number of times. Within this implicit loop there is usually overlap between processor, alignment and memory operations. However, <u>between separate</u> vector instructions, there is no overlap. Thus, one instruction must finish before the next starts. This is what causes the low figures for ELMBAK. This indicates to us that it is important to design the vector instructions and control unit so that different vector instructions overlap each other.

It is also interesting to note that F_P and U_T continue to increase with N for both ELMBAK and SLEQ1. This is due to increased overlap of operations within the implied loops of vector instructions and, in SLEQ1, more efficient recurrence algorithms which are used when N is sufficiently larger than the number of processors.

Table 4.2 indicates the effectiveness of various alignment networks and skewing schemes. As we can see, the crossbar and omega networks performed equally well. The Illiac network performed somewhat better, at least for ADVV and ELMBAK. This is due to two facts. First, the Illiac network was set to operate four times faster than the other networks. This reflects the difference in the complexity of the networks. Second, we were able to "compile" the programs using very simple alignment requirements which could be easily handled by all three networks. The lack of difference between staight storage (0,1) and skewed storage (1,1) is also a reflection of this second point. We were able to compile the programs so they only needed access to rows, and thus they do not benefit from skewed storage. However, we do not believe this result will hold for larger,

Program	Crossbar		Omega		Illiac IV	
	Straight	Skewed	Straight	Skewed	Straight	Skewed
ADVV	1416 (9.7)	1416 (9.7)	1416 (9.7)	1416 (9.7)	1384 (9.9)	1384 (9.9)
ELMBAK	1760 (2.0)	1760 (2.0)	1760 (2.0)	1760 (2.0)	1261 (2.8)	1282 (2.8)
SLEQ1	2644 (4.5)	2644 (4.5)	2644 (4.5)	2644 (4.5)	*	*

*SLEQ1 contains recurrences which we have not yet programmed on Illiac type interconnections.

Table 4.2 Execution time, T_p, and speed factor (F_p) using various alignment networks and skewing schemes. (p=16).

more complicated programs.

Table 4.3 illustrates another interesting result. One question which continually plagues machine designers concerns the relative speed of the memory and processor. Should the memory be the same speed as the processor, twice as fast, or three times as fast? The answer depends on many things: the design of the machine instructions, the size of arithmetic expressions in the source program, etc. Table 4.3 shows the execution time, T_P, and processor utilization, U_T, for three different cases. In column 1, the processor array, alignment network, and memory all have the same cycle time. In column 2, the alignment network alone has been made twice as fast. There is very little difference between columns 1 and 2. This is because the faster crossbar switch is only effective when data alignments are required in the absence of memory accesses. None of these three programs required such alignment. The small diference present between columns 1 and 2 simply represents a shorter overall time for a "short" vector operation in the absence of inter-instruction overlap.

Column 3 of Table 4.3 corresponds to a machine whose alignment network and memories are twice as fast as the processor array. For ADVV, the improvement in T_P is noticeable but not significant. This is because ADVV has relatively large expressions in the source program so the ratio of memory to processor operations is close to 1:1. Thus, ADVV does not need a very fast memory. For ELMBAK and SLEQ1, however, the improvement in T is more significant. This would indicate that, at least for these programs, the faster memory might be cost effective.

Table 4.4 shows the effectiveness of an independent scalar processor and scalar memory on T_P. Also included in the table are the utilizations of scalar memory and scalar processor respectively. A scalar processor and memory should be effective for several reasons. First, without a scalar memory, when a scalar is being broadcast over all elements of an array, the scalar operand would have to be fetched from the array memory and aligned (broadcast). This constitutes wasteful use of the array memory. Second, the use of both scalar memory and processor would allow some scalar operation to be done simultaneously with array operations. Thus we would be able to overlap or mask out certain truckulent serial operations in the program.

In Table 4.4 we can see that the scalar processor causes no improvement in T_P and the scalar memory results in only marginal improvement, even though both are utilized to some extent. However, we believe that our "compiler" can be improved so as to utilize the scalar hardware more effectively. This will involve improving the inter-instruction overlap and more accurate accounting for

Program	Col 1	Col 2	Col 3
ADVV	1416 43%	1384 44%	1036 58%
ELMBAK	1760 6%	1650 7%	1023 10%
SLEQ1	2644 14%	2590 14%	1606 23%

Table 4.3 The effects of the relative differences in memory, alignment, and processor cycle times on T_p and U_T. (16 processors, crossbar switch, N=10, and skewed storage.)

Program	Neither	Scalar memory only	Scalar processor only	Both scalar processor and memory
ADVV	1544 -, -	1416 12%, -	1544 -, 3%	1416 12%, 3%
ELMBAK	1854 -, -	1760 12%, -	1854 -, 12%	1760 12%, 12%
SLEQ 1	2768 -, -	2644 8%, -	2768 -, 9%	2644 8%, 9%

Table 4.4 The effect on execution time, T_p, of a scalar memory and scalar processor. The percentage figures are scalar memory utilization and scalar processor utilization respectively.

such things as subscript calculation.

The above discussion is based only on limited amount of experimental results. It can only be regarded as an illustration of how to interpret the results. According to which "benchmark" programs a user is interested in, he can conduct experiments using that set of programs. In the end, he will then be able to determine on the kind of machine configurations that is most suitable for him.

5. CONCLUSION

This thesis concerns the utilization and effectiveness of interprocessor connection networks for parallel (SIMD) type computers. The problems concerning interconnection networks can be divided into three areas: capabilities, exploitation, and effectiveness.

Capabilities include network properties and network control methods. One of the networks that we have examined closely is the omega network. The omega network is one of the more attractive multistage networks. It is moderate in network complexity and quite powerful in its permutation capabilities. If we concentrate on only some of the more common permutations, we can further reduce the complexity of its control algorithms. Three different control methods are shown in this thesis. We have discussed a significant new property of the omega network, the partitioning property. We showed that a large size omega network can be regarded as a conglomeration of many smaller size omega networks, each passing a different smaller omega-passable connection function. This partitioning property of the omega network proves to be vital for the efficient handling of many computation algorithms. We also discussed another important property of the omega network, the broadcasting ability. We showed the conditions under which a 2-dimensional data array can be broadcast to a 3-dimensional data array using the omega network. This data transfer ability is necessary for example in certain matrix multiplication and recurrence solving algorithms. In Chapter 2, we were also able to extend the capabilities of the omega network further using the concept of linear permutations.

The shuffle connection is the basis of many interconnection networks, like the omega network, the Batcher network, and the binary Benes network. Because of this similarity, we can apply some of the properties of one network to the others, and hence increase further understanding of such networks. Some such extensions are shown in Section 2.3 and 2.4.

Because of the great simplicity in gate counts, the one stage perfect shuffle exchange network is also carefully examined. Algorithms were presented in Section 2.5 to show how such network can be used for performing permutations.

With the old and new knowledge we have acquired on network capabilities, we would like to apply them to some common computations. Recurrence solving algorithms and matrix multiplication algorithms are two examples that we used in Chapter 3. The efficient handling of recurrence operations is essential in parallel processing systems because the parallel system will be degraded to a serial machine otherwise. With careful planning, we were able to simplify the alignment requirements of various recurrence solving algorithms. So, instead of a full crossbar, we can now use a simple alignment network, such as the Omega network. In Section 3.3, we show how a common computation algorithm (such as matrix multiplication), if detected, can be adapted onto a parallel processing system equipped with only a one stage network. Hence Chapter 3 has been dedicated to techniques for exploiting various interconnection networks.

To evaluate the true effectiveness of a particular interconnection network, we have to determine the effectiveness on real programs of a parallel processing system equipped with such a network. This can be achieved with the help of the Analyzer/Simulator project currently being developed. The program analyzer first generates a highly parallelized version of the program. Then the RRG will compile it into suitable pseudo machine code from the information about the parallel processing systems that the user defines. This pseudo compilation is done based on the capabilities of the architecture to be studied, including the type of interconnection network used. From the results of the simulator, we can determine how well does an interconnection network work.

With the methodology described in this thesis, the true effectiveness of an interprocessor connection can then be determined.

We conclude this thesis by giving the following topics that are worthwhile for further research:

1) Is there a set of basic permutation patterns for the omega ROM control method from which other useful permutation patterns can be generated by doing logical oepprations on some members of the set? For example, how can we generate the control pattern of a k-shifted p-ordered spread permutation from that of a k-shift permutation and that of a p-ordered permutation?

2) Finding the analytic bounds and averages for the time required to pass a permutation using a one stage perfect shuffle network is a worthwhile project.

3) We are able to adapt recurrence solving algorithms on a parallel Processing System equipped with a (log n)-stage network. A highly significant result would be to show how recurrence algorithms could be handled a one-stage network (perhaps coupled shuffle and shift connections). This would lead to a very cheap and effective prallel architecture.

4) Control unit times should be carefully added to the Simulator. Subscript calculations and register and scalar usages should be accounted for more accurately.

LIST OF REFERENCES

[1] K.J.Thurber,"Interconnection Networks--A Survey and Assessment," AFIPS Conference Proceedings, Vol.43, pp.909-919, May 1974.

[2] K.E.Batcher,"Sorting Networks and Their Applications," Proceedings of the 1968 SJCC,pp.307-314.

[3] V.E.Benes, Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press, New York, 1965.

[4] D.H.Lawrie,"Access and Alignment of Data in an Array Processor," IEEE Transactions on Computers, pp.1145-1155, December 1975.

[5] T.Feng,"Data Manipulating Functions in Parallel Processors and Their Implementations,"IEEE Transactions on Computers, pp.309-318,March 1974.

[6] G.H.Barnes,et al,"The Illiac IV Computer," IEEE Transactions on Computers,pp.746-757,August 1968.

[7] R.C.Swanson,"Interconnections for Parallel Memories to Unscramble p-ordered Vectors,"IEEE Transactions on Computers, pp.1105-1115, November 1974.

[8] H.S.Stone,"Parallel Processing With the Perfect Shuffle," IEEE Transactions on Computers,pp.153-161, February 1971.

[9] S.W.Golomb,"Permutation by Cutting and Shuffling,"SIAM Review, vol.3,no.4,pp.293-297,October 1961.

[10] M.C.Pease,"An Adaptation of the Fast Fourier Transform to Parallel Processing,"JACM,pp.252-264,April 1968.

[11] L.R.Goke and G.J.Lipovski,"Banyan Networks for Partitioning Multiprocessor Systems,"1st Annual Computer Architecture Conference, Gainsville, Florida, December 1973,pp.21-28.

[12] K.E.Batcher,"The Multi-Dimensional Access Memory in STARAN," submitted to IEEETC.

[13] K.Y.Wen and D.H.Lawrie,"Omega Network Control Plane Implementation," Dept. of Computer Science, Univ. of Illinois, Urbana-Champaign, unpublished memo, Sept. 1976.

[14] D.H.Lawrie,"Memory-Processor Connection Networks," Univ. of Illinois, Urbana-Champaign, Computer Science Report 557, Feb. 1973.

[15] T.Lang and H.S.Stone,"A Shuffle-Exchange Network with Simplified Control,"IEEE Transactions on Computers, pp.55-65,January 1976.

[16] M.C.Pease,"The Indirect Binary n-Cube Microprocessor Array," submitted for publication.

[17] D.C.Opferman and N.T.Tsao-Wu,"On a Class of Rearrangeable Switching Networks," Bell System Technical Journal, vol.50, pp.1579-1618,May-June 1971.

[18] T.Lang,"Interconnections Between Processors and Memory Modules Using the Shuffle-Exchange Network," IEEE Transactions on Computers,pp.496-503, May 1976.

[19] P.Budnik and D.J.Kuck,"The Organization and Use of Parallel Memories," IEEE Transactions on Computers, pp.1566-1569, December 1971.

[20] P.M.Kogge and H.S.Stone,"A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," IEEE Transactions on Computers,pp.786-792, August 1973.

[21] D.Heller,"On the Efficient Computation of Recurrence Relations," Inst. Comput. Appl. Sci. Eng.(ICASE), June 1974.

[22] S.C.Chen and D.J.Kuck,"Time and Parallel Processor Bounds for Linear Recurrence Systems,"IEEE Transactions on Computers, pp.701-717,July 1975.

[23] S.c.Chen,D.J.Kuck,and A.H.Sameh,"Practical Parallel Triangular System Solvers," in preparation.

[24] A.H.Sameh and R.P.Brent,"Solving Triangular Systems on a Parallel Computer," Univ. of Illinois, Urbana-

Champaign, Computer Science Report 766, November 1975.

[25] S.C.Chen,"Speedup of Iterative Programs in Multiprocessing Systems," Univ. of Illinois, Urbana-Champaign, Computer Science Report 694, Jan. 1975.

[26] B.R.Leasure,"Compiling Serial Languages for Parallel Machines," M.S.Thesis,University of Illinois,1976.

[27] S.E.Orcutt,"Implementation of Permutation Functions in ILLIAC IV-Type Computers,"IEEE Transactions on Computers, pp.929-936, September 1976.

APPENDIX A

We will first show a multiplication algorithm that takes N processors and 2N memories. The skewing scheme is $\sqrt{N}+1$-skew 2-skip. Memories 2i and $(2i+N+\sqrt{N}+1)$ mod(2N) are connected to processor i. All even RA's refer to memory 2i and all odd RA's refer to memory $(2i+N+\sqrt{N}+1)$mod(2N). An illustration of the memory map is shown in Figure A.1.

P0		P1		P2		P3	
M0	M7	M2	M1	M4	M3	M6	M5
(0,0)		(0,1)		(0,2)		(0,3)	
	(1,2)		(1,3)		(1,0)		(1,1)
(2,1)		(2,2)		(2,3)		(2,0)	
	(3,3)		(3,0)		(3,1)		(3,2)

Figure A.1 $\sqrt{N}+1$-Skew 2-Skip Scheme

Algorithm:

```
A. Repeat for IC=0 to N-1;
   1. Fetch A, RA=IC, into R1.
   2. Set k=[ ( N+1)*IC ]mod(2N).
   3. If k is odd then k=(k-N- N-1)mod(2N).
   4. k=k/2.
   5. IR=(PPN-k)modN.
   6. Repeat for IT=0 to N-1;
      a. fetch B, RA=IR, into R2.
```

```
        b. multiply R1 and R2 into R3.
        c. G-permute IR.
        d. J=(PPN-( N+1) IR/2 )modN.
        e. if IR is odd then J=(J+N/2)modN.
        f. store R3 into T, RA=J.
        g. G-permute R1.
    7. Set R1=C.
    8. Repeat for IR=0 to N-1;
        a. fetch T, RA=IR, into R2.
        b. add R1 and R2 into R1.
        c. G-permute R1.
        d. G-permute IR.
    9. Store R1 into C, RA=IC.

B. Done.
```

Another algorithm also uses 2N memories, except now it uses a 1-skew 2-skip storage scheme.

For this scheme, memories 2i and (2i+N+1)mod(2N) are connected to processor i. The memory scheme is illustrated in Figure A.2.

P0		P1		P2		P3	
M0	M5	M2	M7	M4	M1	M6	M3
(0,0)		(0,1)		(0,2)		(0,3)	
	(1,2)		(1,3)		(1,0)		(1,1)
(2,3)		(2,0)		(2,1)		(2,2)	
	(3,1)		(3,2)		(3,3)		(3,0)

Figure A.2 1-Skew 2-Skip Scheme

Algorithm:

```
A. Repeat for IC=0 to N-1;
   1. Fetch A, RA=IC, into R1.
   2. If IC is odd then k=(IC-N-1)mod(2N).
   3. k=k/2.
   4. IB=(PPN-k)modN.
   5. Repeat for IT=0 to N-1;
      a. fetch B, RA=IB, into R2.
      b. multiply R1 and R2 into R3.
      c. G-permute IB.
      d. J=(PPN- IB/2 )modN.
      e. if IB is odd then J=(J+N/2)modN.
      f. store R3 into T, RA=J.
      g. G-permute R1.
   6. Set R1=0.
   7. Repeat for IT=0 to N-1;
      a. fetch T, RA=IB, into RA.
```

```
        b. add R1 and R2 into R1.

        c. G-permute R1.

        d. G-permute IR.

    8. Store R1 into C, RA=IC.

B. Done.
```

APPENDIX B

The following twelve tables are the experimental results of three Fortran Programs: ADVV, ELMBAK, and SLEQ1.

The column AN shows what alignment network is chosen. The column (M=) indicates whether the number of menories is prime or not. The column SKEW shows the skewing scheme being chosen. Finally, the column SPEED(P/A/M) gives the relative speeds of processor, alignment network and memory. As for the headings, the number of processors is given as P=xx. SP/SM indicates whether the switches SP and SM (c.f. Section 4.2.2) are turned on or not.

For the tabale on Execution Time(T_p) and Speed Factor(F_p), the main number in each entry is T_p and the number in parentheses is F_p. The remaining tables show utilization measures of various system resources. They are in the order: array memory, input alignment network, output alignment network, vector processor system, scalar memory and scalar processor.

When the entries in some row j is the same as that of row i, it will be marked "same as row i".

PROG: ADVV

N: 10

	AN	M =	SKEW	SPEED (P/A/M)	P = 4 SP/SM=(0,0)	P = 16 (0,0)	P = 16 (0,1)	P = 16 (1,0)	P = 16 (1,1)	P = 64 SP/SM=(1,1)
1	XB	prime	(1,1)	8/8/8	4272 (3.2)	1544 (8.9)	1416 (9.7)	1544 (8.9)	1416 (9.7)	1416 (9.7)
2				8/4/8	4240 (3.25)	1520 (9.05)	1384 (9.9)	1520 (9.1)	1384 (9.9)	1384 (9.9)
3				8/4/4	3012 (3.26)	1088 (8.9)	1036 (9.4)	1072 (9.1)	1036 (9.4)	1036 (9.4)
4		2^k	(1,1)	8/8/8		same	as	row	1	
5				8/4/4		same	as	row	3	
6			(0,1)	8/8/8		same	as	row	1	
7	Ω	2^k	(1,1)	8/8/8		same	as	row	1	
8				8/4/4		same	as	row	3	
9	I4	2^k	(1,1)	8/2/8	4240 (3.25)	1520 (9.1)	1384 (9.9)	1520 (9.1)	1384 (9.9)	1392 (9.88)
10				8/1/4	2997 (3.24)	1077 (9.1)	1032 (9.4)	1069 (9.1)	1032 (9.4)	1032 (9.4)
11			(0,1)	8/2/8	4240 (3.3)	1520 (9.1)	1384 (9.9)	1520 (9.1)	1384 (9.9)	1384 (9.9)
12				8/1/4	2997 (3.2)	1077 (9.1)	1032 (9.4)	1069 (9.1)	1032 (9.4)	1032 (9.4)

Figure A.3 Execution Time(T_p) and Speed Factor(F_p) For Program ADVV

	AN	M =	SKEW	SPEED (P/A/M)	P = 4	P = 16				P = 64
					SP/SM=(0,0)	(0,0)	(0,1)	(1,0)	(1,1)	SP/SM=(1,1)
1	XB	prime	(1,1)	8/8/8	81,74, 7 68, -, -	78,71, 8 65, -, -	73,68, 6 71,12, -	78,71, 8 62, -, 3	73,68, 6 68,12, 3	73,68, 6 68,12, 3
2				8/4/8	81,37, 3 69, -, -	80,36, 4 66, -, -	75,35, 3 72,12, -	80,36, 4 63, -, 3	75,35, 3 69,12, 3	75,35, 3 69,12, 3
3				8/4/4	57,53, 5 97, -, -	56,50, 6 92, -, -	50,46, 4 97, 8, -	56,51, 6 90, -, 4	50,46, 4 93, 8, 4	73,68, 6 93, 8, 4
4		2^k	(1,1)	8/8/8	81,74, 7 68, -, -	78,71, 8 65, -, -	73,68, 6 71,12, -	78,71, 8 62, -, 3	73,68, 6 68,12, 3	73,68, 6 68,12, 3
5				8/4/4		same	as	row	3	
6			(0,1)	8/8/8		same	as	row	1	
7	Ω	2^k	(1,1)	8/8/8		same	as	row	1	
8				8/4/4		same	as	row	3	
9	I4	2^k	(1,1)	8/2/8	81, 8, - 69, -, -	80,33, - 66, -, -	75, 33, - 72,12, -	80,33, - 63, -, 3	75,33, - 69,12, 3	75,63, - 69,12, 3
10				8/1/4	58, 5, - 97, -, -	56,23, - 93, -, -	50,22, - 97, 8, -	57,24, - 90, -, 4	50,22, - 93, 8, 4	50,43, - 93, 8, 4
11			(0,1)	8/2/2	81, 2, - 69, -, -	80, 3, - 66, -, -	75, 0, - 72,12, -	80, 3, - 63, -, 3	75, 0, - 69,12, 3	75, 0, - 69,12, 3
12				8/1/4	58, 1, - 97, -, -	56, 2, - 93, -, -	50, 0, - 97, 8, -	57, 2, - 90, -, 4	50, 0, - 93, 8, 4	50, 0, - 93, 8, 4

Figure A.4   Duty Cycle(U_a) of Various Resources For Program ADVV

PROG: ADVV

N: 10

				SPEED	P = 4	P = 16				P = 64
	AN	M =	SKEW	(P/A/M)	SP/SM=(0,0)	(0,0)	(0,1)	(1,0)	(1,1)	SP/SM=(1,1)
1	XB	prime	(1,1)	8/8/8	62,62,59 82, -, -	52,53,41 61, -, -	59,59,59 61,100, -	52,53,41 63,-,100	59,59,59 63,100,100	15,15,15 16,100,100
2				8/4/8		same	as	row	1	
3				8/4/4		same	as	row	1	
4		2^k	(1,1)	8/8/8	77,78,75 82, -, -	55,56,44 61, -, -	63,63,63 61,100, -	55,56,44 63,-,100	63,63,63 63,100,100	16,16,16 16,100,100
5				8/4/4		same	as	row	4	
6			(0,1)	8/8/8		same	as	row	4	
7	Ω	2^k	(1,1)	8/8/8		same	as	row	4	
8				8/4/4		same	as	row	4	
9	I4	2^k	(1,1)	8/2/8	77,68, - 82, -, -	55,58, - 61, -, -	63,63, - 61,100,-	55,58, - 63,-,100	63,63, - 63,100,100	16,16, - 16,100,100
10				8/1/4		same	as	row	9	
11			(0,1)	8/2/8	77,25, - 82, -, -	55, 6, - 61, -, -	63, 0, - 61,100,-	55, 6, - 63,-,100	63, 0, - 63,100,100	16, 0, - 16,100,100
12				8/1/4		same	as	row	11	

Figure A.5 Slicing Utilization(U_s) of Various Resources for Program ADVV

N: 10

	AN	M =	SKEW	SPEED (P/A/M)	P = 4	P = 16				P = 64
					SP/SM=(0,0)	(0,0)	(0,1)	(1,0)	(1,1)	SP/SM=(1,1)
1	XB	prime	(1,1)	8/8/8	50,46, 4 56, -, -	40,37, 3 39, -, -	43,40, 3 43,12, -	40,37, 3 39, -, 3	43,40, 3 43,12, 3	11,10, 1 11,12, 3
2				8/4/8	50,23, 2 57, -,-	41,19, 2 40, -, -	44,20, 2 44,12, -	41,19, 2 40, -, 3	44,20, 2 44,12, 3	11, 5, 4 11,12, 3
3				8/4/4	35,33, 3 80, -, -	29,26, 2 56, -, -	30,27, 2 59, 8, -	29,28, 2 56, -, 4	30,27, 2 58, 8, 4	8, 7, 1 15, 8, 4
4		2^k	(1,1)	8/8/8	63,58, 5 56, -, -	43,40, 3 39, -, -	46,43, 4 43,12, -	43,40, 3 39, -, 3	46,43, 4 43,12, 3	12,11, 1 11,12, 3
5				8/4/4	44,41, 3 80, -, -	31,28, 2 56, -, -	32,29, 2 59, 8, -	31,29, 3 56, -, 4	32,29, 2 58, 8, 4	8, 7, 1 15, 8, 4
6			(0,1)	8/8/8		same	as	row	4	
7	Ω	2^k	(1,1)	8/8/8		same	as	row	4	
8				8/4/4	44,41, 3 80, -, -	31,28, 2 56, -, -	32,29, 2 59, 8, -	31,29, 2 56, -, 4	32,29, 3 58, 8, 4	8, 7, 1 15, 8, 4
9	I4	2^k	(1,1)	8/2/8	63, 5, - 57, -, -	44,19, - 40, -, -	47,21, - 44,12, -	44,20, - 40, -, 3	47,21, - 44,12, 3	12,10, - 11,12, 3
10				8/1/4	45, 4, - 80, -, -	31,14, - 56, -, -	32,14, - 59, 8, -	31,14, - 57, -, 4	32,14, - 59, 8, 4	8, 7, - 15, 8, 4
11			(0,1)	8/2/8	63, 5, - 57, -, -	44, 0, - 40, -, -	47, 0, - 44,12, -	44, 2, - 40, -, 3	47, 0, - 44,12, 3	12, 0, - 11,12, 3
12				8/1/4	45, 3, - 80, -, -	56, 2, - 93, -, -	32, 0, - 59, 8, -	31, 1, - 57, -, 4	32, 0, - 59, 8, 4	8, 0, - 15, 8, 4

Figure A.6 Combined Utilization(U_T) of Various Resources for Program ADVV

PROG: ELMBAK
N: 10

	AN	M =	SKEW	SPEED	P = 4	P = 16				P = 64
				(P/A/M)	SP/SM=(0,0)	(0,0)	(0,1)	(1,0)	(1,1)	SP/SM=(1,1)
1	XB	prime	(1,1)	8/8/8	2200 (1,61)	1854 (1.91)	1760 (2.01)	1854 (1.91)	1760 (2.01)	1760 (2.01)
2				8/4/8	1900 (1.86)	1552 (2.28)	1450 (2.44)	1552 (2.28)	1450 (2.44)	1450 (2.44)
3				8/4/4	1383 (1.62)	1050 (2.13)	1023 (2.19)	1050 (2.13)	1023 (2.19)	1023 (2.19)
4		2^k	(1,1)	8/8/8		same	as	row	1	
5				8/4/4		same	as	row	3	
6			(0,1)	8/8/8		same	as	row	1	
7	Ω	2^k	(1,1)	8/8/8		same	as	row	1	
8				8/4/4		same	as	row	3	
9	I4	2^k	(1,1)	8/2/8	1735 (2.03)	1392 (2.54)	1282 (2.76)	1392 (2.54)	1282 (2.76)	1280 (2.76)
10				8/1/4	1095 (2.05)	804 (2.79)	774 (2.89)	804 (2.79)	774 (2.89)	772 (2.90)
11			(0,1)	8/2/8	1795 (1.97)	1371 (2.58)	1261 (2.80)	1371 (2.58)	1261 (2.80)	1277 (2.77)
12				8/1/4	1125 (1.99)	793 (2.82)	763 (2.94)	793 (2.82)	763 (2.94)	771 (2.91)

Figure A.7 Execution Time(T_p) and Speed Factor(F_p) For Program ELMBAK

N: 10

	AN	M =	SKEW	SPEED (P/A/M)	P = 4 SP/SM=(0,0)	P = 16 (0,0)	P = 16 (0,1)	P = 16 (1,0)	P = 16 (1,1)	P = 64 SP/SM=(1,1)
1	XB	prime	(1,1)	8/8/8	72,44,27 32, -, -	58,34,24 20, -, -	49,26,23 22,12, -	58,34,24 9, -,12	49,26,23 9,12,12	49,26,23 9,12,12
2				8/4/8	83,26,16 37, -, -	69,20,14 24, -, -	59,16,14 26,15, -	69,20,14 11, -,14	59,16,14 11,15,15	59,16,14 11,15,15
3				8/4/4	57,35,22 51, -, -	51,30,21 36, -, -	42,22,20 37,11, -	51,30,21 16, -,20	42,22,20 16,11,21	42,22,20 16,11,21
4		2^k	(1,1)	8/8/8		same	as	row	1	
5				8/4/4		same	as	row	3	
6			(0,1)	8/8/8		same	as	row	1	
7	Ω	2^k	(1,1)	8/8/8		same	as	row	1	
8				8/4/4		same	as	row	3	
9	I4	2^k	(1,1)	8/2/8	91,13, - 41, -, -	77,26, - 27, -, -	67,24, - 30,17, -	77,26, - 12, -,15	67,24, - 13,17,17	67,27, - 13,17,17
10				8/1/4	72,10, - 65, -, -	67,22, - 47, -, -	55,20, - 49,14, -	67,22, - 21, -,27	55,20, - 21,14,28	55,22, - 21,14,28
11			(0,1)	8/2/8	88,22, - 39, -, -	78,22, - 28, -, -	68,19, - 30,17, -	78,22, - 12, -,16	68,19, - 13,17,17	67,23, - 13,17,17
12				8/1/4	70,18, - 63, -, -	68,19, - 48, -, -	56,16, - 50,14, -	68,19, - 21, -,27	56,16, - 22,14,28	55,19, - 21,14,28

Program A.8 Duty Cycle(U_a) of Various Resources For Program ELMBAK

PROG: ELMBAK

N: 10

				SPEED	P = 4	P = 16				P = 64
	AN	M =	SKEW	(P/A/M)	SP/SM=(0,0)	(0,0)	(0,1)	(1,0)	(1,1)	SP/SM=(1,1)
1	XB	prime	(1,1)	8/8/8	41,43,39 65, -, -	17,19,15 31, -, -	20,24,16 31,100,-	17,19,15 63,-,100	20,24,16 63,100,100	5, 6, 4 16,100,100
2				8/4/8		same	as	row	1	
3				8/4/4		same	as	row	1	
4		2^k	(1,1)	8/8/8	52,54,49 65, -,-	19,21,17 31, -, -	22,26,17 31,100,-	19,21,17 63,-,100	22,26,18 63,100,100	5, 6, 4 16,100,100
5				8/4/4		same	as	row	4	
6			(0,1)	8/8/8		same	as	row	4	
7	Ω	2^k	(1,1)	8/8/8		same	as	row	4	
8				8/4/4		same	as	row	4	
9	I4	2^k	(1,1)	8/2/8	52,28, - 65, -, -	19,30, - 31, -, -	22,34, - 31,100,-	19,30, - 63,-,100	22,34, - 63,100,100	5, 9 ,- 16,100,100
10				8/1/4		same	as	row	9	
11			(0,1)	8/2/8	52,52, - 65, -, -	19,30, - 31, -, -	22,27, - 31,100,-	19,23, - 63,-,100	22,27, - 63,100,100	5, 8, - 16,100,100
12				8/1/4		same	as	row	11	

Figure A.9 Slicing Utilization(U_s) of Various Resources For Program ELMBAK

N: 10

	AN	M =	SKEW	SPEED (P/A/M)	P = 4 SP/SM=(0,0)	P = 16 (0,0)	P = 16 (0,1)	P = 16 (1,0)	P = 16 (1,1)	P = 64 SP/SM=(1,1)
1	XB	prime	(1,1)	8/8/8	30,19,11 21, -, -	10, 6, 4 6, -, -	10, 6, 4 7,12, -	10, 6, 4 6, -,12	10, 6, 4 6,12,12	2, 2, 1 2,12,12
2				8/4/8	34,11, 6 24, -, -	12, 4, 2 8, -, -	12, 4, 2 8,15, -	12, 4, 2 7, -,14	12, 4, 2 7,15,15	3, 1, 1 2,15,15
3				8/4/4	24,15, 8 33, -, -	9, 6, 3 11, -, -	9, 5, 3 11,11, -	9, 6, 3 10, -,20	9, 5, 3 10,11,21	2, 1, 1 3,11,21
4		2^k	(1,1)	8/8/8	37,24,13 21, -, -	11, 7, 4 6, -, -	11, 7, 4 7,12, -	11, 7, 4 6, -,12	11, 7, 4 6,12,12	3, 2, 1 2,12,12
5				8/4/4	30,19,11 33, -, -	10, 6, 4 11, -, -	9, 6, 4 11,11, -	10, 6, 4 10, -,20	9, 6, 3 10,11,21	2, 1, 1 3,11,21
6			(0,1)	8/8/8		same	as	row	4	
7	Ω	2^k	(1,1)	8/8/8		same	as	row	4	
8				8/4/4		same	as	row	5	
9	I4	2^k	(1,1)	8/2/8	47, 4, - 27, -, -	15, 8, - 8, -, -	15, 8, - 9, 17, -	15, 8, - 7, -,15	15, 8, - 8,17,17	4, 2, - 2,17,17
10				8/1/4	37, 3, - 42, -, -	13, 7, - 14, -, -	12, 7, - 15,14, -	13, 7, - 13, -,27	12, 7, - 13,14,28	3, 2, - 3,14,28
11			(0,1)	8/2/8	45,12, - 26, -, -	15, 5, - 9, -, -	15, 5, - 9,17, -	15, 5, - 8, -,16	15, 5, - 8,17,17	4, 2, - 2,17,17
12				8/1/4	36, 9, - 41, -, -	13, 4, - 15, -, -	12, 4, - 15,14, -	13, 4, - 13, -,27	12, 4, - 14,14,28	3, 2, - 3,14,28

Figure A.10 Combined Utilization(U_T) of Various Resources For Program ELMBAK

PROG: SLEQ1

N: 10

	AN	M =	SKEW	SPEED (P/A/M)	P = 4 SP/SM=(0,0)	P = 16 (0,0)	(0,1)	(1,0)	(1,1)	P = 64 SP/SM=(1,1)
1	XB	prime	(1,1)	8/8/8	5577 (2.2)	2768 (4.3)	2644 (4.5)	2768 (4.3)	2644 (4.5)	1284 (9.3)
2				8/4/8	5405 (2.2)	2732 (4.4)	2590 (4.6)	2732 (4.4)	2590 (4.6)	1230 (9.7)
3				8/4/4	3131 (2.0)	1628 (3.9)	1568 (4.1)	1628 (3.9)	1606 (4.1)	1086 (5.9)
4		2^k	(1,1)	8/8/8	5577 (2.2)	2768 (4.3)	2644 (4.5)	2768 (4.3)	2644 (4.5)	2644 (4.5)
5				8/4/4	3131 (2.0)	1628 (3.9)	1568 (4.1)	1628 (3.9)	1606 (4.1)	1606 (4.1)
6			(0,1)	8/8/8		same	as	row	4	
7				8/4/4		same	as	row	5	
8	Ω	2^k	(1,1)	8/8/8		same	as	row	4	
9				8/4/8	5405 (2.2)	2732 (4.4)	2590 (4.6)	2732 (4.4)	2590 (4.6)	2590 (4.6)
10				8/4/4		same	as	row	5	
11			(0,1)	8/8/8		same	as	row	4	
12				8/4/4		same	as	row	5	

Figure A.11 Execution Time(T_p) and Speed Factor(F_p) For Program SLEQ1

PROG: SLEQ1
N: 10

	AN	M	SKEW	SPEED	P = 4	P = 16				P = 64
				(P/A/M)	SP/SM=(0,0)	(0,0)	(0,1)	(1,0)	(1,1)	SP/SM~(1,1)
1	XB	prime	(1,1)	8/8/8	90,63,24 43, -, -	96,66,28 46, -, -	92,62,29 48, 8, -	95,66,28 37, -, 9	92,62,29 38, 8,10	46,22,22 79,17,20
2				8/4/8	93,33,13 45, -, -	97,34,14 46, -, -	94,32,15 48, 8, -	97,34,14 37, -, 9	94,32,15 39, 8,10	48,12,11 82,17,21
3				8/4/4	80,56,22 77, -, -	81,56,24 78, -, -	77,53,24 81, 7, -	81,56,24 62, -, 16	76,51,24 63, 7,16	27,13,13 93,10,23
4		2^k	(1,1)	8/8/8	90,63,24 43, -, -	96,66,28 46, -, -	92,62,29 48, 8, -	95,66,28 37, -, 9	92,62,29 38, 8,10	92,62,29 38, 8,10
5				8/4/4	80,56,22 77, -, -	81,56,24 78, -, -	77,53,24 81, 7, -	81,56,24 62, -,16	76,51,24 63, 7, 16	76,51,24 63, 7,16
6			(0,1)	8/8/8		same	as	row	4	
7				8/4/4		same	as	row	5	
8	Ω	2^k	(1,1)	8/8/8		same	as	row	4	
9				8/4/8	93,33,13 45, -, -	97,34,14 46, -, -	94,32,15 48, 8, -	97,34,14 37, -, 9	94,32,15 39, 8,10	94,32,15 39, 8,10
10				8/4/4		same	as	row	5	
11			(0,1)	8/8/8		same	as	row	4	
12				8/4/4		same	as	row	5	

Program A.12 Duty Cycle(U_a) of Various Resources For Program SLEQ1

PROG: SLEQ1

N: 10

	AN	M =	SKEW	SPEED (P/A/M)	P = 4 SP/SM=(0,0)	P = 16 (0,0)	P = 16 (0,1)	P = 16 (1,0)	P = 16 (1,1)	P = 64 SP/SM=(1,1)
1	XB	prime	(1,1)	8/8/8	43,64,45 62, -, -	21,46,22 30, -, -	23,51,23 30,100,-	21,46,22 36,-,100	23,51,23 36,100,100	11,23, 3 12,100,100
2				8/4/8		same	as	row	1	
3				8/4/4		same	as	row	1	
4		2^k	(1,1)	8/8/8	53,72,57 62, -, -	22,47,24 30, -, -	24,52,24 30,100,-	22,47,24 36,-,100	24,52,24 36,100,100	8,38, 6 12,100,100
5				8/4/4		same	as	row	4	
6			(0,1)	8/8/8	49,74,59 62, -, -	20,49,26 30, -, -	22,54,26 30,100,-	20,49,26 36,-,100	22,54,20 36,100,100	7,41,10 12,100,100
7				8/4/4		same	as	row	6	
8	Ω	2^k	(1,1)	8/8/8		same	as	row	4	
9				8/4/8		same	as	row	4	
10				8/4/4		same	as	row	4	
11			(0,1)	8/8/8		same	as	row	6	
12				8/4/4		same	as	row	6	

Figure A.13 Slicing Utilization(U_s) of Various Resources For Program SLEQ1

PROG: SLEQ1
N: 10

	AN	M =	SKEW	SPEED	P = 4	P = 16				P = 64
				(P/A/M)	SP/SM=(0,0)	(0,0)	(0,1)	(1,0)	(1,1)	SP/SM=(1,1)
1	XB	prime	(1,1)	8/8/8	39,41,11 27, -, -	20,31, 6 14, -, -	21,32, 6 14, 8, -	20,31, 6 13, -, 9	21,32, 6 14, 8,10	5, 5, 1 9,17,20
2				8/4/8	38,21, 6 28, -, -	20,15, 3 14, -, -	21,16, 3 14, 8, -	20,15, 3 13, -, 9	21,16, 3 14, 8,10	5, 3, - 9,17,21
3				8/4/4	34,36,10 48, -, -	17,26, 5 23, -, -	17,26, 5 24, 7, -	17,26, 5 22, -,16	17,26, 5 22, 7,16	3, 3, - 11,10,23
4		2^k	(1,1)	8/8/8	47,46,14 27, -, -	21,31, 7 14, -, -	22,32, 7 14, 8, -	21,31, 7 13, -, 9	22,32, 7 14, 8,10	7,23, 2 4, 8,10
5				8/4/4	42,41,12 48, -, -	18,27, 6 23, -, -	19,27, 6 24, 7, -	18,27, 6 22, -,16	18,27, 6 22, 7,16	6,19, 1 7, 7,16
6			(0,1)	8/8/8	44,47,14 27, -, -	20,33, 7 14, -, -	20,34, 8 14, 8, -	20,33, 7 13, -, 9	20,34, 8 14, 8,10	6,26, 3 4, 8,10
7				8/4/4	40,42,13 48, -, -	17,28, 6 23, -, -	17,28, 6 23, 7, -	17,28, 6 22, -,16	16,28, 6 22, 7,16	5,21, 2 7, 7,16
8	Ω	2^k	(1,1)	8/8/8		same	as	row	4	
9				8/4/8	49,24, 7 28, -, -	22,16, 3 14, -, -	22,16, 4 15, 8, -	22,16, 3 13, -, 9	22,17, 4 14, 8,10	7,12, 1 4, 8,10
10				8/4/4		same	as	row	5	
11			(0,1)	8/8/8		same	as	row	6	
12				8/4/4		same	as	row	7	

Figure A.14 Combined Utilization(U_T) of Various Resources For Program SLEQ1

VITA

Kuo Yen Wen was born in Shanghai, China on November 14, 1949. He received his B.S. degree in Electrical Engineering and Computer Science in 1971, and his M.S. degree in Computer Science in 1974, both from University of Illinois, Urbana-Champaign. From 1971 to 1976, he was a research assistant in the Department of Computer Science. He was associated with the Illiac III project from 1971 to 1973 and with the Machine and Software Organization project since 1973. He was the coauthor with Prof. Duncan H. Lawrie of a paper, "Effectiveness of Various Processor/Memory Interconnections," presented at the 1976 International Conference on Parallel Processing. He is a member of the Institute of Electrical and Electronic Engineers.

IIOGRAPHIC DATA IIT	1. Report No. UIUCDCS-R-76-830	2.	3. Recipient's Accession No.

le and Subtitle Interprocessor Connections-- Capabilities, Exploitation and Effectiveness	5. Report Date October, 1976
	6.
thor(s) io Yen Wen	8. Performing Organization Rept. No. UIUCDCS-R-76-830
rforming Organization Name and Address iiversity of Illinois at Urbana-Champaign partment of Computer Science bana, Illinois 61801	10. Project/Task/Work Unit No.
	11. Contract/Grant No. US NSF MCS73-07980 A03
onsoring Organization Name and Address itional Science Foundation ishington, D. C.	13. Type of Report & Period Covered Doctoral Dissertation
	14.

. applementary Notes

. bstracts

icently, some research interests has centered around interprocessor connections for MD type parallel machines. However, we still lack a methodology for evaluating irious networks. In this paper, we first present some new results on network roperties. Then we show how to exploit various networks in ordinary computations. nally we describe how we can apply the theoretical results to predict the rformance of some network in a real program environment, which is the true measure network effectiveness.

. ey Words and Document Analysis. 17a. Descriptors

ray Slicing
mputation Adaptation
twork Control
rmutation Capabilities
MD Type Machine
mulation

7b dentifiers/Open-Ended Terms

7c COSATI Field/Group

3. ailability Statement lease Unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 176
	20. Security Class (This Page UNCLASSIFIED	22. Price